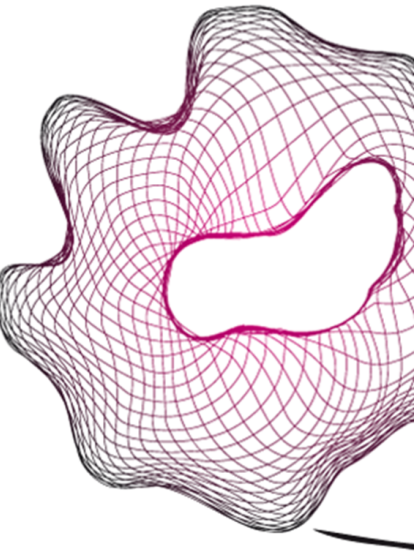
UNIVERSITY OF TWENTE.



Faculty of Electrical Engineering, Mathematics & Computer Science

End-to-end Encrypted Data in Web Applications

Milo Cesar

MSc Thesis
November 2021

Graduation Committee:
dr.ing. F.W. Hahn
dr. M. Daneva
dr. T. van Dijk MSc
ir. P.R. Heuver

Services and CyberSecurity
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

End-to-end encryption is often considered to be the holy grail of encryption, at this time, however, it is not a common feature of web applications. Encryption can be used to increase both the privacy and security of users. The web has seen a great move towards more encryption in recent years, as can be seen by the adoption of TLS. TLS, however, does not provide end-to-end encryption, it encrypts data between the user and a server, not between end-users. While this is a great step forward, it should not be the endpoint of this movement towards a more encrypted web. The next logical step is to move towards encryption between end-users, the so-called end-to-end encryption. This research created a software design for a software system that, when implemented, can make end-to-end encryption obtainable for most web applications. Our software design is created to work in a multi-user environment, it outlines how to create a nested authentication system, how to distribute keys, and how to revoke access. Our research also discusses technical difficulties that are encountered when implementing our software design, and it shows that such a software system can be implemented by web application with a negligible impact in terms of run-time overhead. We performed benchmarks on three JavaScript crypto libraries to measure this run-time overhead. These benchmarks showed us that hundreds of encryption operations can be performed without impacting the user experience. Furthermore, this research shows a design for this software system that allows web application developers to extend their applications with very little effort. This all comes together to form a solid basis from which a system can be implemented that will increase the security and privacy of many web application users, while at the same time staying easy for developers to implement.

Contents

Chapter 1

Introduction

The security of the internet is a topic that is discussed every day. Oftentimes these talks are related to encryption. On the one hand, tech platforms are calling for more encryption, on the other hand, the legislative branch is calling for required back-doors in encryption [1]. At the moment, companies seem to be winning this fight - as evident from the fact that more and more services are increasing their encryption usages. Examples of which are the popular instant messaging app, Whatsapp, who introduced end-to-end encryption in 2016 [2], one year later DropBox introduced a partnership with "BoxCryptor" to introduce end-to-end encryption in its cloud storage solution [3]. The video conference system Zoom has introduced end-to-end encryption in 2020 [4] after receiving a complaint from the Federal Trade Commission of the United States for claiming the presence of this feature while "Zoom did not provide end-to-end encryption for any Zoom meeting" [5].

The resources that large co-operations have at their disposal, enable them to ensure proper encryption of the data they are handling. This does not hold for smaller businesses that lack extensive development teams with the required cryptography knowledge. This is problematic since the benefits of encryption are not exclusive to these large co-operations. Almost every digital service can benefit from the increased security that encryption can provide. Take for example the many data breaches, both large and small, that occur at an alarming rate. In many cases, encryption can severely reduce the impact of such a data breach by limiting the amount of useful information that an attacker can obtain.

Many consider end-to-end encryption to be the holy grail in encryption. Properly implemented, end-to-end encryption will result in only the end-users being able to decrypt any data transmitted via the service. This protects you not only against attackers, but also against the company maintaining the service, the hosting company providing the servers, and state agents such as intelligence agencies that might try to breach your privacy.

Somebody knowledgeable in cryptography or cybersecurity might be able to create a solution for your encryption needs fairly easily. When the needs and economic incentives are large enough, almost any problem can be solved with enough time and resources. Large companies such as Facebook can spend the money to integrate end-to-end encryption in their apps as is evident when looking at WhatsApp. But what about smaller companies, with smaller development teams? Understandably, developing new features can be preferable over implementing end-to-end encryption, and even if they wanted to implement any form of encryption, many companies lack the expertise necessary to implement it.

To summarize, there exists a need for end-to-end encryption in many applications but there is no easy and affordable solution to gain the desired properties.

To this avail, RiskChallenger[1] has asked us to explore the possibilities of developing a software solution that introduces end-to-end encryption in a plug-and-play type fashion. They would like to see a software solution that could transform any JavaScript web application into an end-to-end encrypted application. Such a solution should manage the keys, the encryption algorithms, and the transfer of the data to the storage solution.

[1] `https://riskchallenger.nl`

To create this plug-and-play type library we aimed to create a library based on the aspect-oriented programming paradigm. Decorators would be used in a class to annotate properties that need encryption. In the simplest form, this would allow us to encrypt a property by first defining a key, for example when deriving a key from a username and secret by using `deriveKey(username, secret)`, followed by marking the to-be-encrypted property with the `@Encrypt` decorator and pointing it to the key-holding property of the class. Listings 1.1 and 1.2 show the changes that should be required in this simplest form.

```
class User {
    public username: string;

    public personalData: string;
}
```

Listing 1.1: User class with unencrypted personal data

```
class User {
    public username: string;

    @Encrypt('key')
    public personalData: string;

    public key = deriveKey(username, secret);
}
```

Listing 1.2: User class with encrypted personal data

Implementing this software system turned out not to be feasible in the given time frame while taking into consideration the constraints with regards to the technologies that were to be used. This research, therefore, focused on providing the information that will help RiskChallenger or other researchers who might continue to implement this software solution.

This research's main contributions are as follows:

- We created a software design for a software system that can add end-to-end encryption to an existing system while requiring minimal code changes to do so
- We answered common design and implementation-specific questions, such as how to distribute and revoke keys, which will be encountered when implementing end-to-end encryption
- We identified and explained technical difficulties that must be carefully considered before practical implementation.
- We benchmarked three cryptography libraries which showed that minimal overhead, with regards to page load times, can be expected when adding end-to-end encryption to a system

Chapter 4 shows the software design which conforms to the wishes of RiskChallenger, chapter 5 answers questions that have been asked by RiskChallenger and which will naturally arise when implementing this software system. Chapter 6 outlines implementation-specific problems that arose when we were implementing the software system. Lastly, chapter 7 will contain the benchmark of three JavaScript-based cryptography libraries to compare their speeds and come to an expected overhead with regards to page load times.

Chapter 2

Background

As stated in the previous section, there are many reasons why one would like to use encryption. However, the costs associated with encryption, both monetarily and functionally, are high. These costs might not be feasible for smaller companies, or individuals, to bear. To reduce the monetary costs associated with increasing the use of encryption, this research aimed to develop a plug-and-play type solution that enables some form of encryption for an application where desired.

When designing any security function you will first need to decide what you want to protect against. What are the specific threats? Who are the threat actors? Based on these threats you can outline your requirements, and only then is it useful to start thinking about the details such as what data you want to encrypt, when you want to encrypt it, and what encryption algorithms you want to use.

2.1 Scope

Even before we can look at our threats, we need to set some boundaries. It is not feasible for this project to protect against all possible threats against software. This section, therefore, defines some boundaries which together define the scope of this project. It will outline what aspects of the software we will ignore, what consequences this has, and why we believe that this is a valid boundary.

2.1.1 Trust

An important principle in cybersecurity is the principle of least privilege[6], this entails that every user or account should only have the bare minimum amount of privilege with which it can still properly execute its tasks. This translates to the trust domain very well. In the same way that a user should have the bare minimum amount of privileges, our system should have to trust as little as possible, both in the way of humans as well as systems.

To this avail, the software that this research aimed to produce will trust the end-user, their system, their browser, and later to be defined software.

The first of which is a requirement for the software to work. At some point, somebody will have to be able to use the data for the data to be of any use. This also entails an assumption on the honesty of the user of the data. This software will not protect against the willful extraction of information by the user.

The second entry, being the system belonging to the user, is trusted from a feasibility standpoint, it is tremendously hard to validate the security of any such system. This includes both the hardware as well as the software of the system. Many threats can compromise the security of an application without the ability for the application to reasonably protect against these threats. Take for example a key-logger that might be installed on the system of a user. One way to protect against this is to only allow input from a custom UI element in your application which represents a keyboard, this would negatively impact the user experience to an extreme extend. Another example might be a compromised

graphics card that relays all data it renders to an external server. While this might sound extreme, Bloomberg has made claims of Chinese state-sponsored organizations compromising hardware in US logistic chains [7]. The claims made by Bloomberg have been unambiguously denied by all involved entities. However, a recent update on this story has Bloomberg reconfirm its allegations with new sources and information which indicates that the attack is still ongoing[8]. The only fail-proof way to protect against these kinds of attacks is by handcrafting all hardware components.

The third entry, which is the browser, is also trusted from a feasibility standpoint, since this system is designed for web applications, it is not feasible to protect against the platform they run on. To ensure the security of a browser, one can look through the source code if this is available and afterward confirm that the browser is indeed a compiled form of the source code that was previously checked. This, however, is something that needs to be done by the end-user. The four largest browsers (Chrome, Safari, Edge, and Firefox), which combine to see over 90% of the browsers market-share[9] do not provide any functionality for websites to validate the integrity of the browser.

The last entry, being later to be defined software, is intentionally vague to facilitate change when it is deemed necessary. At the very least this category will encompass the `SubtleCrypto`[1] library which is JavaScript's default crypto library, this must be trusted since creating or implementing your own cryptography library is very error-prone. In the words of Bruce Schneider "Creating a cipher is easy. Analyzing it is hard." [10]. A crypto library, however, is not the only software we need to trust. Any JavaScript library, that is included in the web page which uses this software system, can access all rendered data. It is, therefore, necessary that we trust these included libraries. Furthermore, the software that is used to distribute our webpage needs to be trusted to serve the correct web page. Subresource integrity[11] will help to some extend here. It can be used to validate the integrity of another resource that will be loaded by the webpage by checking a hash of the resource content against the hash that was originally provided to the web page for this resource. However, the first page which includes these other resources must then be trusted since its hashes could also be modified by an attacker. Once again this also includes trust in the browser, since it also has to properly validate these hashes.

In the end, it is important to be knowledgeable about which parties you need to trust. When this is documented, it is easy for each user to draw their conclusions on whom they want to trust, and thus, by extension, if they can trust this software.

2.1.2 CIA triad

The CIA triad[2] consists of three foundational security principles.

Confidentiality
: This ensures that only authorized persons can access the data. This principle is commonly violated through lack of access control or data breaches.

Integrity
: This ensures that the data is correct, its values are as intended. This principle could be violated through lacking form validation or data corruption.

[1] `https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto`

[2] The exact origin of the term 'CIA triad' is unknown. The underlying concepts were already in use in military contexts millennia ago as evident in the works of Julius Caesar[12]

Availability
This ensures that the data is available at the necessary times. This principle could very well be violated in the case of a DDOS attack or through ransomware.

As outlined in chapter 1 the primary goal of this project is to protect the confidentiality of the data, this primary focus stems from the desires of the client for their use cases. Per best practices, the data that is entrusted to RiskChallenger is backed up redundantly. The reason that this backup strategy can be used to remedy violations of the integrity of the data as well as many violations of the availability of the data. The confidentiality of data, however, can not be restored once it is violated, it is therefore of paramount interest to protect the confidentiality. Logically, any increase in security in the other facets of this triad is welcomed but this research will not claim to increase the security in any facet but the confidentiality.

2.1.3 Time of encryption

"The three states of information"[13] are the three broad time categories at which we can utilize encryption, each of which brings their challenges and have their specific benefits.

In Transit
Whenever data is moving between two points. Most commonly between a client and a server but this could also be between two servers, two clients, or a plethora of other options. Most commonly this transit would depend on infrastructure under the control of other entities such as network cables and switches. At this stage, encryption is most commonly used to prevent man-in-the-middle attacks.

In Use
Whenever data is actively in use. This might be in the case of processing the data on a server or when displaying the data to a client. The usability of encryption depends heavily on the specific use case. Showing encrypted data to a user is of no use to that user. However, a server might be able to work with encrypted data, for example, it might be able to use homomorphic encryption to perform operations on encrypted data without ever gaining knowledge of the data that was encrypted. Gaining access to data that is currently in use turns out to be hard. In most cases, this would require modification to the source code of the software or access to the hardware that is running the software to extract data from its memory.

At Rest
Whenever the data is in storage. This is mostly understood to be stored on any non-volatile medium, such as, but not limited to, storage in databases, on a hard disk drive, or on tape. Using encryption at this time can prevent the biggest data leaks, most applications store more data than they are actively using at any time.

Based on the assumed principle of least trust, we need to ensure the security of the data at all times. One of these time categories is already covered when assuming the scope as defined until this point. Data in transit is commonly protected through TLS, over the HTTPS protocol. The encryption and decryption of data in transit are covered by the browser and the server which distributes the webpage. Two entities which we trust based on section 2.1.1. Google's latest reports indicate that at least

97% of the browsing time is performed over the HTTPS protocol[14]. We will therefore assume that TLS is used to protect the data between the client and the server. We furthermore assume that the TLS connection is configured according to best practices and that certificates which are issued for this purpose are valid. These assumptions lead to a system in which the security of the data can be guaranteed while in transit.

2.1.4 Metadata

Metadata is a broad category of data. The oxford dictionary describes it as "information that describes other information in order to help you understand or use it". In web applications, this includes access patterns or relations between data. Hiding metadata is outside of the scope of this project. Some of the research that we will discuss in chapter 3 does hide this metadata, their research shows that hiding this is tremendously difficult. Furthermore, the applications that RiskChallenger develops do not contain confidential metadata.

2.2 Functional requirements

While the previous section outlined the scope and with that said what we can ignore, the following section will define some of the functional requirements of the system which require special attention. This could be because they are often overlooked by other research, due to their importance to the client, or due to their difficulty.

2.2.1 Multi-user access

The client requires that multiple users must be able to read and/or write data. For example, when storing an arbitrary unit of data, multiple users must be able to read from it and write to it. The users that can read certain data must not necessarily be able to write to the same data. A system that allows for nesting is desirable, meaning that multiple users can belong to a group, after which this group can be given read and/or write access, which implies that the users belonging to the group have the same access level. Successful implementation of this paves the way to role-based access control (RBAC) which gives developers a lot of flexibility in defining their access control.

Multi-user access brings multiple complications with it, the most important of which is key management. How do you distribute the keys of a new user to the other users of the system while ensuring the integrity and confidentiality of said key?

2.2.2 Access revocation

Access- and by extension key-revocation is an often ignored topic in this field of research as will become evident when we are discussing the related work in chapter 3. However, it is as important, or even more so, as the step of introducing new users. While this research assumes that the user of the system is to-be-trusted, this trust can be broken. It is easy to understand why companies would want to revoke access to data for an ex-employee. This assumption on the user's trustworthiness implies

that the user did not copy or alter any data in the system. This trust however is no longer present after termination. Depending on the situation it might be desirable to remove access to all current and previous data, however, in all cases, we need to ensure that the ex-employee can not see any new updates on the data.

These requirements will make the aforementioned key-management issues even harder. Any key that could have been accessed by this now not-to-be-trusted user, must be cycled, and all other users who need continued access to the encrypted data must be notified of the new keys.

2.3 Threats

As became clear in section 2.1, our primary target is to protect the confidentiality of the data while in use or in transit from everybody but the intended reader. With this in mind, there are two primary threats to encrypted data, theft of the encrypted data itself and theft of the encryption keys. Both of which are necessary to get to the un-encrypted data

2.3.1 Theft of encrypted data

A data leak through any means, i.e. an SQL injection error in the web app, wrong session management that allows a user to see the data of another user, or a curious database administrator, will break our confidentiality requirement. To protect the data at this moment, it will be encrypted. Depending on the level of confidentiality and functional requirements of the app, multiple different types of encryption can be used. Each of these encryption algorithms come with their own security and functionality drawbacks.

2.3.2 Theft of encryption keys

When an adversary obtained the encrypted data, it will also need to get a hold of the encryption keys. Without this the previously stolen data is useless. Only when an adversary obtains both the data and the keys, is their attack succeeded.

2.4 Threat actors

Any application can be interacted with by many actors, some of which might be honest users who need to interact with the application, some of the actors might be curious as to see some data without the intention of harming anybody, and some of the actors might be outright malicious actors that want to see the world burn. We distinguish between passive attackers, who only observe, and active attackers who are willing to alter traffic, data, source code, etc.

The following is a non-exhaustive list of possible threat actors. Each of which brings its own skill-set and privileges. These threat actors will be used in the later stages of this research to show the security of the software solution.

2.4.1 Honest but curious database administrator

The honest but curious database administrator has full privileged access to the database. We assume that it needs this access to perform the work it does for the software owner. The access provided to this individual already leaks the encrypted data to them. They, however, do not need access to the encryption keys, this prevents them from accessing the plain-text data. They will only observe data, making them a passive attacker.

2.4.2 Nefarious software developer

A nefarious software developer can access the source code of the application, it can access the server on which the software is hosted. Their abilities allow them to place a backdoor in the software to listen in on the network traffic after the TLS connection has been terminated. These abilities make them active attackers.

2.4.3 Hacker

The hacker has no elevated access to the infrastructure hosting the application. It has no connection to the company hosting and/or maintaining the service and therefore has less to lose. It is not limited by moral boundaries, it will do anything it can to gain access to the data that is stored in the application. They are an active attacker.

Chapter 3

Related work

This research and its corresponding software solution are not the first to explore end-to-end encryption for web applications. Multiple existing solutions already exist, this section will discuss these existing solutions and their limitations, chapter 8 will give an overview of the differences between the solution as proposed by our reserach and the research that is discussed here.

CryptDB: Protecting Confidentiality with Encrypted Query Processing

In 2011, Popa et al. introduced CryptDB [15], a system which uses a proxy server and "*onions of encryption*" to create an encrypted database. Their system generates session keys for its users. These session keys are derived from the users' passwords. They allow for the nesting of these keys to share data between multiple users, i.e. a forum can have a key to encrypt all data in that forum while the user can use their key to retrieve this forum key. This allows efficient multi-party access to encrypted data.

The onion design of the system allows selectively revealing attributes of the data to only reveal the necessary data. For their order union which, as the name implies, should be used to order columns. They will first encrypt the data with an order-preserving encryption scheme followed by an IND-CPA secure algorithm. Only if the user should be able to access the data, they can remove the outer layer and expose the order-preserving properties.

The protections imposed by CryptDB protect against 2 main threats. The first is a curious database administrator, an entity that does not alter any queries or results. This might include a compromised server. CryptDB tries to hide the data from this entity, metadata such as the number of entries in a table or the table structure will not be hidden. Neither will the actual queries. The second threat is a general 'arbitrary threat' this mainly focuses on threats where the proxy server is compromised. This might lead to the leaking of encryption keys. CryptDB does not prevent this, it merely tries to reduce damages by using different keys for different data. This ensures that only actively accessed data is leaked in this case.

An important shortcoming of CryptDB is that it has no special consideration for key revocation. It does not describe nor has special support for revoking users' access to data.

Building web applications on top of encrypted data using Mylar

In continuation of their research into encrypted data for web applications, Popa et al. introduced Mylar [16] in 2016. Mylar improves on CryptDB in almost every way. The cost associated with these improvements is relatively little, Mylar requires client-side code changes which CryptDB did not. Mylars researchers claim that Mylar requires an average of 36 lines of code to be implemented in a web application.

Mylar's most important change to protect against passive attackers is related to the point of decryption. While CryptDB uses its server-side proxy to decrypt the data, Mylar decrypts the data on the

client-side web application.

Passive attackers are not the only threat considered in Mylar, it also aims to protect against active attackers in multiple ways. Mylar uses a browser plugin, in combination with multiple root domains and content security policies to ensure the authenticity of the client-side code through code-signing. This prevents an attacker from distributing fake versions of the web app. This protection does require that the user validates the URL on which it is about to enter information; it cannot protect against phishing attacks.

Mylar moves the burden of account management to an identity provider. This identity provider should keep track of the public key of each user. This ensures that an active attacker could not perform a man-in-the-middle attack on keys whenever a user wants to share their information with another user. This requires that the active attacker does not have access to the identity provider.

Lastly, Mylar allows for multi-key search, the specific implementation of which is described by Popa et al. in 2013 [17]. It allows efficient search over text that is encrypted using multiple different keys. Other database encryption techniques often require that the to-be-searched text is encrypted under a single key, an assumption that is no longer necessary. This drastically improves the efficiency of searching through multiple text entries which are shared with multiple users.

Its main shortcoming is the same as CryptDB's in that it does not have special consideration for key revocation.

Breaking Web Applications Built On Top of Encrypted Data

The security claims that are guaranteed by Mylar are challenged by Grubbs et al. [18]. In their 2016 paper, partially in a response to Mylar but their techniques can be more broadly applied to find flaws in "client-server applications that aim to hide sensitive user data from untrusted servers".

They found that much metadata could be leaked whenever a snapshot of the database was downloaded. Most of this information is leaked due to misuse of the Mylar system. Too little data was protected or wrong implementations have been used. A persistent passive attacker who can intercept encrypted traffic and get multiple snapshots of the database could leak all search queries due to an error in the multi-key search algorithm. An active attacker could use the previously mentioned problem with the multi-key search algorithm to perform "a brute-force dictionary attack on any past, present, or future search query of any server-hosted content".

A response to these claims has been made by Mylars authors [19] in which they state that the first two described attacks are considered outside of the scope of the application since they rely on metadata which is intentionally leaked by Mylar to preserve performance, while the last attack on the multi-key search algorithm is already described by the Mylar authors in their original paper[16] which relies on an error from either the programmer implementing the system or from the identity provider.

Verena: End-to-End Integrity Protection for Web Applications

Verena [20] once more shows the evolution of these end-to-end encryption systems. It uses the notion of a 'trust context' (TC) to manage the access control for its entities. These TCs are combined with

'Integrity Query Prototypes' (IQP) to specify that "a certain set of read-queries run in a certain trust context", thus limiting the access of the data to the correct users. Its biggest contribution to this set of systems is the security with regards to integrity, while its predecessors had strong guarantees with regards to confidentiality, Verena extends this with strong guarantees to the integrity of the data. It guarantees that "the result of a read query (...) that corresponds to an IQP with a trust context (...) reflects a correct computation on the complete and up-to-date data, ...".

Verena once more ignores the problems imposed by key revocation.

Arx: An Encrypted Database using Semantically Secure Encryption

In direct comparison to CryptDB (and under supervision of one of CryptDBs authors) Arx: An Encrypted Database using Semantically Secure Encryption was developed[21]. It claims that "CryptDB's order and equality queries via PPE schemes are faster than Arx's (...) but also significantly less secure". It uses the same proxy-server tactic as introduced by CryptDB. Its main security benefit over CryptDB is that Arx "incurs pay-as-you-go information leakage". Where CryptDB would leak the frequency count or order relations for every value in the database, Arx limits this to only the data involved in the queries it observes during an active attack. Both of these systems state an overhead in the order of 10% on-target applications.

Database encryption using asymmetric keys: a case study

In their 2017 study, Boicea et al. [22] compared the functionalities and speed of three encryption schemes commonly used in database encryption, those being: RSA, ElGamal, and ECIES. They tested these three algorithms on three distinct key sizes and while manipulating strings of three different lengths. These tests were performed for encryption as well as decryption. Their research showed RSA being the fasted and ECIES being the slowest for encryption with differences increasing with key size and string length. The inverse is true for decryption where ECIES was the fasted and RSA the slowest, once more the differences increased with key size and string length.

These experiments were performed on a relational database, MySQL to be precise, with its built-in encryption system. There was one encryption key for the entirety of the database.

The encryption times lay between 2.40 ms and 69.60 ms. The decryption times were around 3 orders of magnitude smaller, laying between 0.009 ms and 0.076 ms.

The paper also touches on the use of symmetric- and asymmetric algorithms stating that symmetric keys are generally faster and asymmetric keys are generally more secure. They discuss role-based access control but do not deviate on this further, while discussing this they also shortly mention the problems of key distribution while not discussing this further - key distribution is of no concern for their approach since they only use a single key.

End-to-End Encryption Schemes for Online Social Networks

Schillinger and Schindelhauer talk about end-to-end encryption in social networks in their 2019 paper [23]. Their work starts by defining multiple attack models. Like the threat models as used by

Popa et al., these define specific threats, their actors, and their target. The threats they define are focused on the confidentiality and integrity of the available data.

They outline the usage of symmetric- and asymmetric- encryption algorithms to create the desired security and performance. When creating a new chat room, the creator retrieves the public key from all the participants, it uses these keys to encrypt a newly generated symmetric key that can be used to encrypt the messages. While functional, this system is fairly rudimentary in its setup. It fails to account for any form of forward secrecy or key revocation, two properties that are desirable in social networks.

Chapter 4

Software design

The development of a software solution should start with a theoretical solution. This will help the software developer in efficiently creating their software as well as help spot problems early on. In this instance we wanted to create such a theoretical solution for the application that is created by RiskChallenger, conforming to the requirements and wishes that are outlined in chapter 2. The following sections will detail the working of this proposed theoretical solution.

4.1 System Design

The application manages data, all data belongs to exactly one group. Every group has users, users can be part of multiple groups.

Let data be denoted by D_x, let a group with id i be denoted by G_i, and let a user with id i be denoted by U_i. Let a public key be denoted by pk_x, let private keys be denoted by sk_x, let symmetric keys be denoted by k_x, and let derived symmetric keys be denoted by dk_x where x is the owner of the key i.e. pk_{U_1} would be the public key belonging to the user with id 1.

The user has a unique identifier (such as an id or email address) as well as a secret. The unique identifier and secret are used to derive a symmetric key dk_{U_i}. The secret used to derive this key must not be their password since that is used for authentication purposes and thus known to the server.

The group uses a symmetric key k_{G_i} to encrypt the data D_x that belongs to it as well as the other data related to the group such as its name. The group also contains encrypted versions of its symmetric key, it includes exactly one such key for every user that belongs to the group, this key is encrypted with the respective user's public key. These encrypted symmetric keys are stored with the group on the server. The user furthermore has a public/private keypair pk_{U_i}, sk_{U_i}. Its public key is public knowledge while the private key of its keypair is encrypted with its derived symmetric key. The public key as well as the encrypted private key are stored with the user on the server. The derived key is never stored.

In the following section, $Enc(a, b)$ is used to denote that the key a is used to encrypt the data b, respectively $Dec(a, b)$ denotes that the key a is used to decode the data b. A symmetric key k_x can be used for both encryption as well as decryption such that $Dec(k_x, Enc(k_x, D_x)) = D_x$. A public key pk_x can be used to encrypt data, while its corresponding private key sk_x is necessary to decrypt the data such that $Dec(sk_x, Enc(pk_x, D_x)) = D_x$. The keys used in these operations, therefore, dictate whether a symmetric or asymmetric encryption algorithm is used.

Entity	**Key**	**Notation**	**Stored**	**Use**
User	Derived	dk_{U_i}	✗	Encrypting user's asymmetric key
	Asymmetric	sk_{U_i}, pk_{U_i}	✓	Encrypting groups symmetric key
Group	Symmetric	k_{G_i}	✓	Encrypting data

Table 4.1: Summary of keys

4.1.1 Creating a user

To create a new user U_i, the user first generates a new private key sk_{U_i} and the corresponding public key pk_{U_i}. They derive the symmetric key dk_{U_i} from their unique identifier and secret. They send pk_{U_i} as well as $Enc(dk_{U_i}, sk_{U_i})$ to the server to create their account.

4.1.2 Creating a group

To create a new group G_i, a user generates a new symmetric key k_{G_i}. The user U_i then encrypts k_{G_i} with its own public key pk_{U_i}, $Enc(pk_{U_i}, k_{G_i})$. Any other relevant data of the group, such as its name or creation date, can be encrypted using k_{G_i}. The encrypted symmetric key and all encrypted data can then be transferred to the server.

4.1.3 Adding data to a group

When adding data D_x to group G_i the user U_i uses their username and the secret to derive its derived key dk_{U_i}. It gets its own encrypted private key from the server and decrypt it with this derived key $sk_{U_i} = Dec(dk_{U_i}, Enc(dk_{U_i}, sk_{U_i}))$. It decrypts the groups symmetric key with this private key $k_{G_i} = Dec(sk_{U_i}, Enc(pk_{U_i}, k_{G_i}))$. Once the user has obtained the group's symmetric key, it can use that key to encrypt the data $Enc(k_{G_i}, D_x)$. The encrypted data can then be sent to the server.

4.1.4 Adding a user to a group

When adding a user U_y to a group G_i, the user U_i that wishes to add U_y first needs to follow the steps as outlined in section 4.1.3 to obtain the group's symmetric key k_{G_i}. It can then use the public key of the user who should be added pk_{U_y} to encrypt the group's symmetric key $Enc(pk_{U_y}, k_{G_i})$. The updated group including the new encrypted symmetric key can then be sent to the server.

4.1.5 Removing data from a group

Removing data from a group requires no special considerations. The encrypted data can simply be purged from the server.

4.1.6 Removing a user from a group

Removing a user from a group does require special consideration. This consideration and its rationale are discussed in section 5.1. This section gives two options, either re-encrypting all old data or not cryptographically removing access to all old data.

In either case, a new symmetric group key $k_{G_{i_2}}$ needs to be generated. The public keys of all users that should still have access must be used to encrypt this new symmetric key, ensuring their continued access to the group data.

If all data is to be re-encrypted, that can now be done with the new symmetric group key. After this, the old encryptions of the symmetric group key can be removed.

If the data is not to be re-encrypted, the new symmetric key must be exclusively used for encrypting new data but the existing data can remain as-is. This can for example be done by keeping all encrypted symmetric keys in an array to which you push the new encrypted symmetric key. When encrypting data, care must be taken to always do so with the latest key in the array. When data must be decrypted, the keys can be popped from this array until the matching key is found.

4.1.7 Nested authentication

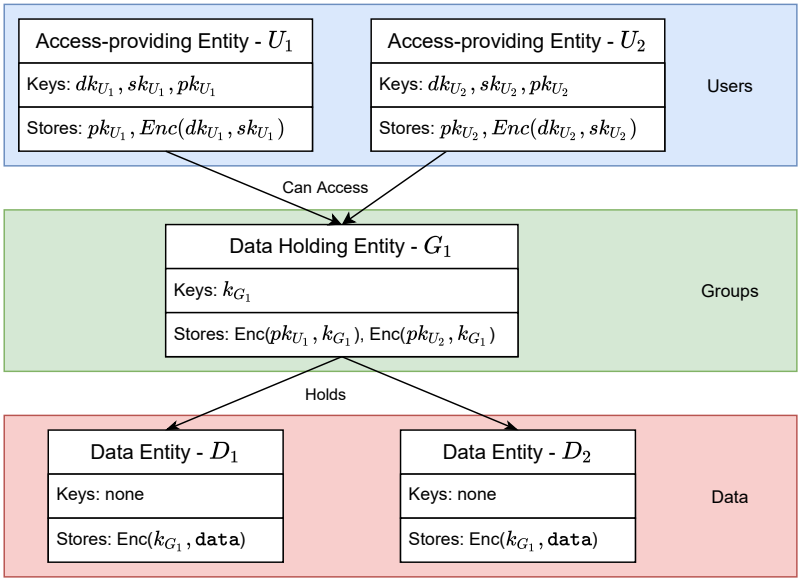


Figure 4.1: Software design with a single access control layer

The system as detailed in section 4.1, illustrated in figure 4.1 has one "layer" of access control, users can access groups, and groups hold data. If a set of people need access to data from multiple groups, each of them will need to be added to each of the groups. Such an access system is common within companies, an example of which might be the executive board of a company wishing access to all the data that their company manages. The designed system can be expanded upon to create such a second access control layer. A company C_i can be created which has a public/private keypair, pk_{C_i} and sk_{C_i}. The company's public key can be used in the same way as a user's public key to encrypt the symmetric key of a group $Enc(pk_{C_i}, k_{G_i})$. To give a user U_i access to company C_i instead of a group, the users public key pk_{U_i} will be used to encrypt the companies private key sk_{C_i}, $Enc(pk_{U_i}, sk_{C_i})$. The user can thereafter obtain the private key of the company $Dec(sk_{U_i}, Enc(pk_{U_i}, sk_{C_i})) = sk_{C_i}$. With the private key of the company, the user can decrypt the group key $Dec(sk_{C_i}, Enc(pk_{C_i}, k_{G_i})) = k_{G_i}$. Doing this does not prevent us from adding users to groups. We now have introduced a second layer of access control to which we can add users, users who are added to a group can utilize all data of said group, while users added to a company can utilize all data that belongs to a group belonging to the company. This principle is illustrated in figure 4.2, here we can see that U_2 can obtain all the keys necessary for the eventual decryption of all data.

This expansion is not limited to adding a single access control layer either, as it turns out, any acyclic

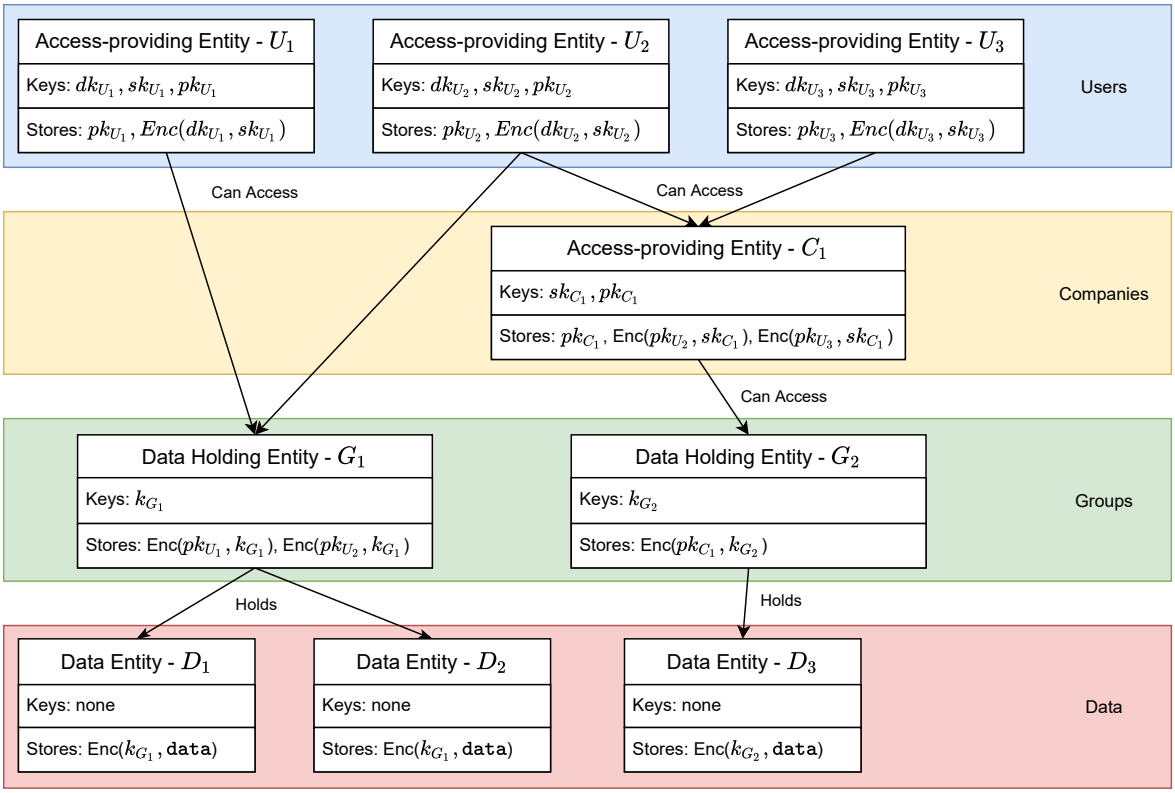


Figure 4.2: Software design with two access control layers

directed graph can be used for authentication purposes in this system. Any entity which has data that needs encryption (a data-holding entity) requires a symmetric key k_x to encrypt this data. Any entity which could be provided access (an access-providing entity) requires a public/private keypair, pk_x and sk_x. When access is provided to a data-holding entity, its symmetric key should be encrypted with the public key of the access-providing entity. When access is provided from access-providing entity x to access-providing entity y, the private key belonging to entity x should be encrypted with the public key of entity y. These two entity types are not mutually exclusive, we could add a symmetric key to a company, who was up until this point exclusively an access-providing entity, of the previous example and perform steps as discussed in section 4.1.3, to give the company the ability to hold data. Making the company both a data-holding as well as an access-providing entity. These principles are illustrated in figure 4.3.

There is no cryptographical limitation for cyclic access control groups, however, they have no practical uses. Any user which has access to an entity in a cycle in the access graph has access to all other entities in the same cycle. Therefore, this cyclic group has the same access level in all entities, thus making it logical to collapse the cycle into a single access control group, reducing the number of cryptographical operations required to manage any data. This shows that not only any *acyclic* directed graph can be used for authentication purposes, this system expands onto any directed graph independent of its cyclicity.

With this nested authentication system, a fine-grained access control system can be put in place.

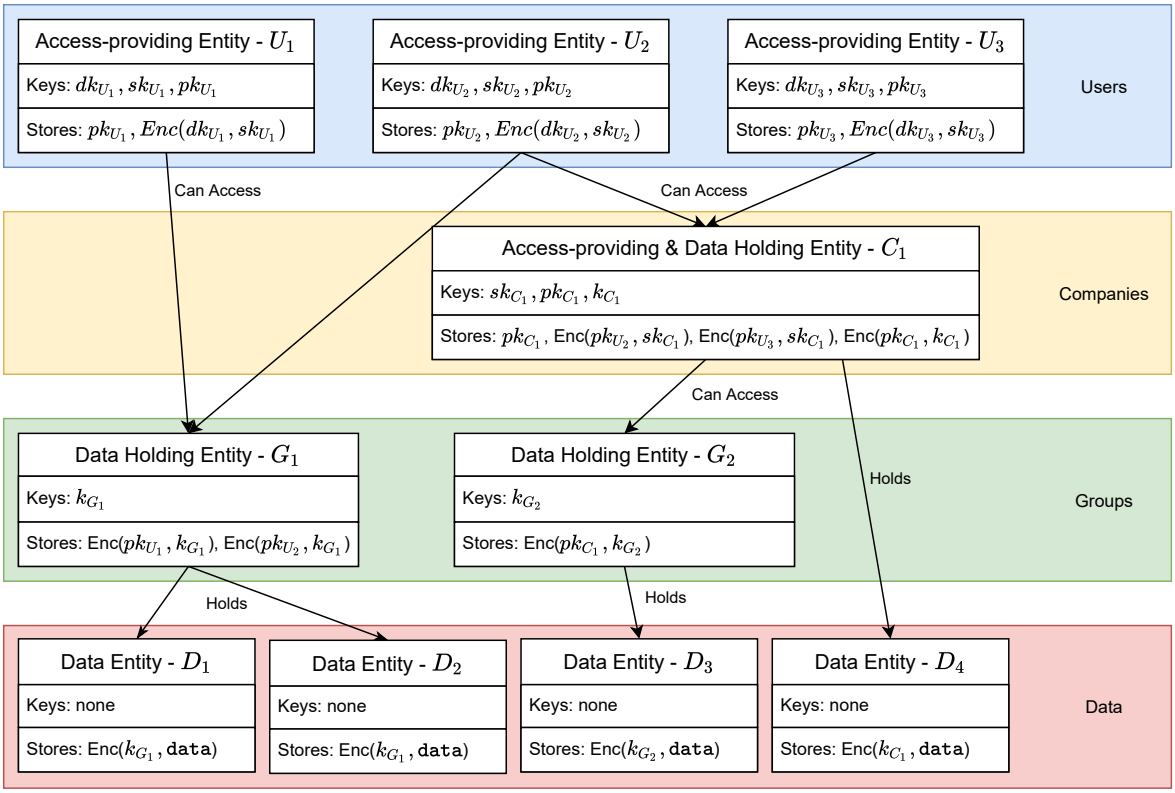


Figure 4.3: Software design with graph-based access control

4.2 Key distribution

In section 4.1.4, it states “use the public key of the user who should be added”, this begs the question that was glossed over in that section: “How do I get this public key?”. In most cases it would be fetched from the server, since it is a public key, the server can freely distribute these keys to whoever might require them. However, this opens up a new attack factor, the public key which is returned from the server might not be the public key from the user that was requested. A bad actor might perform a man-in-the-middle attack in which they intercept the request and return their public keys instead of the requested public key. This attack might lead to the leakage of sensitive data.

The research as discussed in chapter 3 deals with it in multiple ways. Both CryptDB and Arx fail to address the problem, they do not make clear how they retrieve keys from other users [15, 21]. Mylar and Verena state that they use an identity provider to manage the public keys [19, 20], this is a trusted third party, which both the client and server can trust not to be compromised. The last research, talking about online social networks [23] deals with it similarly to Mylar and Verena, by using an identity provider. They expand upon this by loading a certificate, issued by the identity provider, into their applications which they can use to check that the provided public key does indeed come from the identity provider.

Using an identity provider seems the best way to solve this problem. It is important to choose the correct identity provider, it should be a provider which you trust to hand out the correct keys. They should have no affiliation to the app that is to be protected and the developers of the app should not have any influence over the identity provider. The signed keys such as in use by Schillinger et. al [23]

provide some extra security. An example in which this provides extra security is when the DNS of the identity provider gets compromised, allowing legit traffic to the identity provider to end up at the server of a bad actor who could return different keys.

An application might also facilitate checking the fingerprint of a public key to validate the integrity of that key. Each public key has a corresponding fingerprint which is a human-readable representation of the public key. When both parties look at this fingerprint and validate that the fingerprint they have of each other matches the expected value, they can conclude the fingerprint they have of the other party is legitimate. To make this process easier, the fingerprint can be rendered as a QR-code which the other party can scan, this simplifies validating the fingerprint. An example of checking fingerprints through text as well as QR-codes can be seen in the form of Signals "safety number"[24] or Whatsapp "key verification" functionality [25], both of which use the Signal Protocol [26].

4.3 Conformity

The design as outlined in this chapter conforms to the wishes and requirements as outlined in section 2.2. we argue that this design has adequate safeguards in place to protect against the threats as well as the threat actors which have been described in sections 2.3 and 2.4.

4.3.1 Functional requirements

RiskChallenger imposed two functional requirements, multi-user access, and access revocation.

We can look to section 4.1.4 to see an example of multi-user access. The use of public/private key pairs in this context allows for asynchronously adding users to a group. Section 4.1.3 describes how a user who has been added to a group can obtain the symmetric group key. Using this key, every member of the group can access all data belonging to that group. Section 4.1.7 shows how this system can be expanded to include any acyclic access structure, providing great flexibility.

The process of revoking access from a user has been explained in section 4.1.6. It details both, revoking access after which the removed user can still access data it had previous access to, as well as revoking access and switching keys for a possible security increase with a computational overhead. If the client follows either of these steps it will result in a new symmetric group key that the removed user has no access to. When this new key is used to encrypt new data, the removed user can no longer access the new data.

4.3.2 Threats

Chapter 2 outlines two threats and states that a compromise of both the encrypted data and the encryption keys will lead to a compromise of the underlying unencrypted data.

The designed system as outlined in this chapter does not provide any extra protection against theft of the (encrypted) data. It does however have elements that prevent the theft of encryption keys. The proposed design ensures that there is never an unencrypted private or symmetric key on the server, nor will the server have all the required parts to reconstruct any of the derived keys. Chapter 1 states

"... deriving a key from a username and secret ...", section 4.1 is even clearer "This secret must not be their password...", it forbids the use of the user's password for use as this secret. This ensures that at least one part required for deriving the user's symmetric key is always unknown to the server. While the server should never store a user's password in an unhashed fashion, most authentication systems require sending the user's password to the server at some point in the authentication flow so it can be checked against a hashed version of the password. Therefore still "leaking" the password to the server.

4.3.3 Threat actors

The defense against an honest but curious database administrator, who as we can recall from section 2.4 is a passive attacker since they only observe but do not alter data, is fairly straightforward. All sensitive data is encrypted. No plaintext encryption keys are present on the server, all encryption keys which are stored on the server are encrypted. The only key which is not encrypted is the user's derived key. This key is derived from the username, which can be considered known to the database administrator, and a secret, which by its definition from section 4.1 not known to the database administrator. A proper key derivation is based on a hash function, making it unfeasible to retrieve the secret from the key, and is computationally expensive, preventing exhaustive search [27]. The last property prevents the curious database administrator from deriving a user's key and thus keeps sensitive data save from this threat actor.

The nefarious software developer, which we know from section 2.4 to be an active attacker, could get its hand on the password of the users. The previous section outlines how we prevent this access from being a problem for our security model. This design does not fully protect against this threat actor. It has no features to protect the integrity of the web application, a functionality that is present in Mylar and Verena [19, 20]. These checks require additional software to be created and maintained by the developers while also requiring this software to be installed by the end-user. If a nefarious software developer decides to alter the source code of the web application, they could gain access to all data that is accessed while their vulnerable version of the web application is deployed. They could also obtain the derived symmetric keys of any user or the underlying secrets, resulting in access to all data to which the compromised user has access. A user needs to interact with the web application for their data to be compromised. If there is no interaction with the web application while it is compromised, no data will be compromised.

The hacker also has some possibilities to attack the designed system. A hacker might have the possibility to gain themselves the same access as a nefarious software developer through other means, thus resulting in them being able to exploit the same vulnerabilities with the same limitations. They however also have another possible attack. They could gain access to the secret which is used to derive a user's symmetric key through other means such as social engineering. If they obtain such a secret they can reconstruct the user's symmetric key. If they also get their hands on the encrypted data, they could use this key to decrypt the data.

Chapter 5

Library design

While the implementation of the system which aimed to solve RiskChallenger's question, to develop a software solution that introduces end-to-end encryption in a plug-and-play type fashion, was not feasible in the given time-frame, the architecture design is available for possible further continuation of this project. This design was started in chapter 4 with a theoretical model, the current chapter will focus on other key design challenges. It will first discuss the challenges with, and solutions for, revoking access in a multi-user environment. Thereafter it will discuss the best point to insert an encryption library into the lifecycle of an application.

5.1 Revoking access in a multi-user environment

As discussed before in section 2.2, one of the important aspects of end-to-end encrypted applications is access revocation. What happens when a user, who has been previously granted access to information, should no longer have access to this information. As became clear from chapter 3, many research seems to simply ignore this topic [15, 19, 20, 23]. Even real-life implementations, in which removing access is commonplace, do not seem to have any 'smart' implementations for this use case. For example, both the old version [28] and the new version [29] of multi-party implementation for the Signal protocol [26] simply replace the existing group with a new group whenever access for one or more users need to be revoked. The effect of this is that all old communication can still be seen by the removed users while limiting their access to new information. In practice this is a fair trade-off since the removed user has had previous access to this data, therefore they could have already copied this information away. The confidentiality of this specific data is already lost to the specific users.

A more secure option would be to re-encrypt all the old data with the newly generated key. Doing this would prevent the removed user from getting access to the old data if they have not yet copied the data. It comes however at the obvious cost of having to re-encrypt all old messages, depending on the application this could be unbearably slow. This is especially the case when the removed user had access to a large volume of data, every part of which will need to be re-encrypted with the new key and which will need to be sent to the server. Chapter 7 will give more insight into the overhead that can be expected for these re-encrypt operations.

5.2 Library injection point

The goal of this research is to create a software library that enables end-to-end encryption in JavaScript applications. Per definition, this requires all data to be encrypted on the server. At the same time, such an application must be usable, thus requiring that the data can be viewed on a screen. Thus, at some moment after the data reached the client and before it is rendered on the screen, it must be decrypted. This logically raises the question, at what point do you perform this decryption step.

[1]Image licensed under the Apache-2.0 license by Dataroma - `https://github.com/datorama/akita`

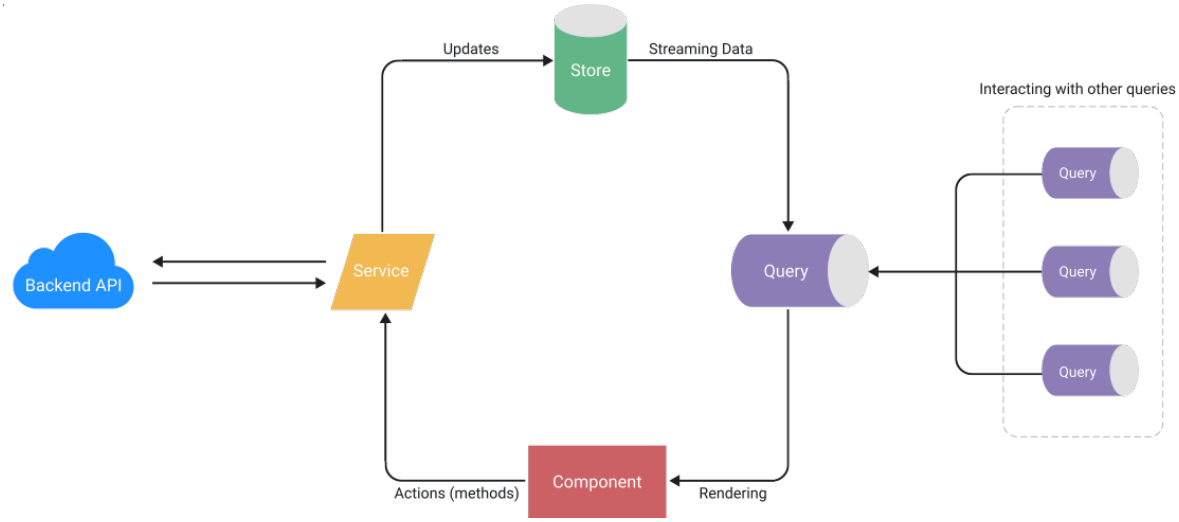


Figure 5.1: Application data life cycle[1]

5.2.1 After data retrieval

If we look at a chronological timeline of data flowing through our application, we would start with the moment our data enters the application. This would correspond to the move from 'Backend API' to 'Service' in figure 5.1. This could be after it is fetched from an API or it could be provided along with the HTML in the server response. In most applications, multiple requests fetch different data, possibly from different sources. For example, in the case of a standard chat application, we could start by loading the authorized user and afterward fetch some groups it belongs to. This incremental approach to fetching data often leads to cases where not all the required information is available. Furthermore, all data is still in its raw form, this severely impacts our ability to modify it. At this stage, the application does not know what class the data belongs to and can therefore not decide which fields require decryption or what keys should be used for this operation. These considerations make decryption at this point impossible.

5.2.2 Before rendering

At the other end of our timeline, the last possible moment to decrypt the data would be before the data is sent to the view and the DOM is modified to show the data. This would correspond to the 'Component' in figure 5.1. At this point, all required data must be available thus decryption must be possible. Performing decryption at this point reduced the amount of time the data is in the application in a decrypted state. At first glance it seems like this reduced the possibility for other libraries, which might interact with this data, to read the data in decrypted form. However, in most cases any JavaScript library that is included in the application has access to the DOM and can thus read the plain text from there, making this point more some sort of security-by-obscurity. For most applications, however, this point would be too late. If the data is rendered at multiple places throughout the applications, the decryption would have to be performed for each of these locations, straining the CPU unnecessarily.

5.2.3 State management

Many applications use some form of state management to keep track of the resources that have been loaded to prevent over-fetching data from a back-end. This can drastically increase the speed of an application while reducing the development time and making it easier to adapt to API changes. Such a state-management library manages the full life cycle of the data. It fetches the data from the API and moves it through its components until a component updates the DOM with the desired value. An overview of the components that the data might flow through is illustrated in figure 5.1, everything between the 'Service' and 'Components' is considered to be part of the state management. The software created by RiskChallenger uses Akita[2] for this purpose.

For the decryption step, we could use the 'Service', 'Store', 'Query' or 'Component'. This is also exactly the order the data moves through from API to view. When new data is created in the 'Component' (i.e. a new message is sent in a group chat or a new user is created) data moves through the 'Component' via the 'Service' to the 'Backend API'. Experience teaches us that it is easiest to keep symmetric operations such as serialization and deserialization, or in this case encryption and decryption in a shared space. This generally leads to reduced code complexity. From our previous observations, we can see that there are two places where the data resides both from the API to the view and the other way around, these would be the 'Component' and the 'Service'. While our previous sections deduced that both just before entering and just after leaving the state-management our bad options for encryption and decryption - it seems that just after entering or just before leaving state-management are the best options.

5.2.4 Best injection point

In the end, we decided that the best moment for decrypting the data is just after the raw data gets transformed into its class representation. Based on our experience, this is the moment that all parent data which could conceivably contain the required keys are loaded. It is also a step where data is manipulated for more accurate or easier to use representation thus combining all these steps nicely. This often maps best to the 'Service' of figure 5.1, as it does for RiskChallengers software. This service would first fetch the data from the API, then deserialize the data, and afterward decrypt the data, to finally push the data to the store.

The same holds for the encryption step. When a component informs the service that data will need to be sent to the API, the service would first encrypt the data, and then serialize the data to finally push it to the API. Specifically, the combination of the encryption with the serialization step is powerful since it allows us to force the removal of unencrypted data before it is sent off to the server.

[2] `https://datorama.github.io/akita/`

Chapter 6

Implementation problems

During the development of any product, it is common to encounter problems. The development of this software product was no different, many problems were encountered, some of which could be fixed, some of which could be bypassed, and some of which required alterations of design choices to accommodate them.

6.1 Algorithm limitations in `SubtleCrypto`

As discussed in section 2.1 it is desirable to trust as few parties as possible while also avoiding having to implement your own cryptography. For a cross-browser compatible web app, this leaves only one source available as an encryption 'library'. This being the built-in encryption of the browser coming in the form of the `SubtleCrypto`[1] JavaScript library. This library is a core part of most modern browsers, 96% of internet users connect to the internet using a browser that includes this library [30]. Since it comes bundled with the browser, we need not trust any more parties with our information. It would be just as easy for the browser to leak the text from the rendered page (or the underlying document) as it is to leak from this encryption library. Most browsers are also backed by large co-operations, providing stronger guarantees about the longevity of the project as well as providing more trust in its implementation since it can be assumed that this implementation has been thoroughly tested. Lastly, these implementations are significantly faster than non-native implementations, test performed in 2017 by the WebKit team show that their native implementation can process files at a speed of up to 1500MB/s while non-native implementations are capped at 16MB/s [31]. Chapter 7 will benchmark 3 common web cryptography libraries to compare their speeds.

Choosing the `SubtleCrypto` encryption library, however, also has its drawbacks. These implementations are often closed source, making it impossible to validate their actual inner workings and prove their security. These large co-operations might also be more susceptible to government interference. Next to the non-technical problems, using this library also introduces technical limitations. The standard defining the `SubtleCrypto` library [32] only requires implementation of the `RSA` and `AES` encryption algorithms. The `RSA` implementation being for `RSA-OAEP` and the `AES` having multiple implementations being `AES-CTR`, `AES-CBC` and `AES-GCM`. The latter set of which once more outlines the need for easier security, first two named implementations for `AES` are vulnerable to a chosen-ciphertext attack[33, 34]and the original fourth implementation of `AES-KW` has been removed from most browsers since it has no known public security proof [35]. Specifically, the CBC mode of operation is known to be vulnerable to a padding attack[36] while the CTR mode of operation is message-malleable[37].

`AES-GCM` is a so-called authenticated encryption scheme. This not only provides confidentiality but also authenticity for the message. This ensures, even before decryption of the message, that the message is (or isn't) encrypted with the expected key and has not been altered. It does this by generating a message authentication code over the encrypted message, this message authentication code in the context of authenticated encryption schemes is called a "tag".

These problems leave us with one usable and secure symmetric encryption algorithm. `AES` through

[1] `https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto`

the `AES-GCM` implementation with a nonce used as IV to create a symmetric non-deterministic authenticated encryption scheme. The `RSA-OAEP` implementation can be used as an asymmetric encryption scheme.

6.2 Text encoding

Most of the operations in the `SubtleCrypto` library use the `Uint8Array` data type. This is an array of unsigned, 8-bit, integers. This is rarely seen in JavaScript since JavaScript is inherently an untyped language, the use of a typed array is therefore often misunderstood by novice developers. The rarity of these `Uint8Array`'s in JavaScript can also be seen by looking at other common JavaScript usages, when deserializing the serialization of an `Uint8Array` the data-type of the returned data is an object instead of an array. Deserialization of the serialization of a JavaScript `Array`, thus being the un-typed variant, will yield data of the expected `Array` type. This provides problems when trying to send any such data to a server or other string-based storage media such as the browser's local storage.

```
1 > JSON.parse(JSON.stringify(new Uint8Array([1, 2, 3])))
2 { '0': 1, '1': 2, '2': 3 }
3 > JSON.parse(JSON.stringify(new Array(1, 2, 3)))
4 [ 1, 2, 3 ]
```

Listing 6.1: Serialization of `Uint8Array`

A commonly used bypass to these problems is the use of text-encoding schemes which turn these integers into strings. It turns out, however, that there are no general-purpose text encoders available for cryptographically secure random data in the standard JavaScript library that comes pre-bundled with the browser. Most available text encoders in this context are used to convert text to binary and back, examples of these include Unicode and ASCII. However, these character encoding schemes are not continuous, meaning that not all integer values can be represented by a valid character in their respective encodings. This leads to un-decodable text due to some values being replaced by an invalid marker. This marker will decode to a different binary value from the originally provided value. Another common text-encoding scheme is base 64, the 64 in its name is a reference to the number of characters in the scheme. These 64 characters can represent 6 bits while our data contains 8-bit integers. Converting to base 64 would therefore either require complicated and hence error-prone bit-level manipulation to fit 3 integers in 4 characters or use 2 characters per integer with at least 4 padded bits per integer.

Our original bypass to this problem was to store these `Uint8Array`'s as comma-separated-values of integers cast to string as shown in listing 6.2. While this is highly ineffective from a storage space perspective, it is dead-simple, fail-proof, and easy to implement. Since this output was stored as a string, each 8-bit integer would on average require 3.57 characters.

```
1 > const data = [106, 19, 233, 207, 175, 108, 24, 33, 71, 157, 26, 88, 129, 208, 185]
2 > data.join(',')
3 "106,19,233,207,175,108,24,33,71,157,26,88,129,208,185"
```

Listing 6.2: Naive character encoding

We then figured out that we could improve on this by using a fixed-width encoding. We assumed that the integer values, which are outputted by the crypto library, are normally distributed. In that case, 156 values out of the 256 values in an 8-bit integer would require 3 characters to represent, take

into consideration the extra comma and we get over an average of 3 characters per integer value. If we were to pad all the integers to 3 characters we can reduce the required storage space to a fixed 3 characters per integer. Since this is a fixed-width encoding, we no longer need to use a separation marker.

```
1 > data.map(entry => entry.padStart(3, '0')).join('')
2 "106019233207175108024033071157026088129208185"
```

Listing 6.3: Fixed-width character encoding

After this, the assumption was that this could still be done more efficiently. Using base 16 encoding, also known as hexadecimal encoding, we can encode all possible 8-bit integer values with 2 characters. Combining this with the 0-padding we got an encoding that is still very simple, does not rely on weird or obscure characters, and is relatively easy to implement. Most important of all, it only uses 2 characters per 8-bit integer.

```
1 > data.map(entry => entry.toString(16).padStart(2, '0')).join('')
2 "6a13e9cfaf6c1821479d1a5881d0b9"
```

Listing 6.4: Fixed-width base 16 character encoding

In the end, we kept a very simple and easy-to-implement character encoding scheme which reduced the needed storage from 3.57 characters per integer to 2 characters per integer, resulting in a 44% reduction of required storage.

Encoding	**Characters**	**Bits**	**Bit-level overhead**
Base 64 per character	2	12	50%
Base 64 bit-joined	1.33	8	0%
Comma separated integers	3.57	28.6	257%
Padded integer join	3	24	100%
Hexadecimal	2	8	0%

Table 6.1: Efficiency of different character encoding schemes.

Table 6.1 compares the possible different text-encoding schemes discussed in this section. It shows how many characters are in use for each integer value and how many bits would be in use if the most efficient encoding is used to store this data. In the case of base 64 and hexadecimal, this would be a binary storage solution. This data shows that two optimal solutions require no overhead over the uncompressed data in binary storage. The hexadecimal encoding requires no further bit-level manipulation making it easier to implement the solution.

6.3 Decorators

JavaScript has the notion of `Decorators`[2], as already used in listing 1.2 in the form of the `@Encrypt()` decorator. They have first been introduced by the babel transpiler in 2015[3]. Since then support for them has increased throughout the JavaScript ecosystem. They are an experimental TypeScript feature[4] and are proposed for JavaScript, awaiting approval to join the JavaScript standard library.

[2] `https://github.com/tc39/proposal-decorators`
[3] `https://github.com/loganfsmyth/babel-plugin-transform-decorators-legacy/commit/ae054b`
[4] `https://www.typescriptlang.org/docs/handbook/decorators.html`

Since decorators are not currently in the JavaScript standard library, they require that the code is transpiled before it can be used in a browser. The fact that there are competing standards from Babel and TypeScript makes development using this functionality difficult, many sources such as blog posts fail to properly disclose which specific type of decorator they are using, leading to obscure bugs triggered by implementation-specific details.

For all their flaws, decorators are still very useful. Popular libraries such as class transformer[5] or class validator[6] make heavy use of them. Their main benefit is that they can be additively implemented in the code. They do not require modifying existing code, they can merely be appended to existing code to provide the extra functionality, following the aspect-oriented design principle.

As we can see from listing 1.2, marking a field for encryption requires one added line of code, namely `@Encrypt('key')`, if we where not to use the decorators this would require overwriting the properties getters and setters as seen in listing 6.5. It would also require the introduction of an extra property to store the decrypted value. As stated in chapter 1, it is RiskChallenger's wish to create an encryption library that is easy to use, requiring overwriting all encrypted properties in existing classes does not fit our definition of easy.

```
1  class User {
2      public username: string;
3
4      private decryptedPersonalData: string;
5
6      public get personalData(): string {
7        return this.decryptedPersonalData;
8      }
9      public set personalData(newValue: string): void {
10       this.decryptedPersonalData = decrypt(newValue, 'key')
11     }
12
13     public key = deriveKey(username, password);
14 }
```

Listing 6.5: User class with encrypted personal data without decorator

6.3.1 Decorator factories

One of the first problems we encountered while implementing decorators is their lack of support for arguments. A decorator is simply another JavaScript function that is called in a specific way, this being the `@` syntax. The `@Encrypt` decorator was to be implemented as a property-decorator, this type of decorator retrieves two parameters. The target, being the class that has the property which has been annotated, and the property key, being the name of the property which has been annotated. To bypass this limitation, decorator factories are used. Decorator factories are functions that accept parameters and which return a decorator function. The lexical scoping of JavaScript allows the decorator function to access the parameters that were supplied to the decorator function. This principle is shown in listing 6.6. When calling this decorator using `@Encrypt('key')` the `Encrypt('key')` is executed first, the result from this function call is then treated as a decorator due to the `@` prefix.

```
1  function Encrypt(keyBearingProperty: string) {
2    return function (target: Function, propertyKey: string) {
```

[5] https://github.com/typestack/class-transformer
[6] https://github.com/typestack/class-validator

```
3       //Decorator implementation
4     }
5   }
```

Listing 6.6: Decorator factory implementation

6.3.2 Property definition

The getters and setters as shown in listing 6.5 are required for the operation of the library, it has to know when data is retrieved from the API and set on the class instance so it can decrypt the data. Setting such properties at runtime can be done with JavaScript using the `Object.defineProperty`[7] method.

```
1 Object.defineProperty(target, propertyKey, {
2   set: function(newValue: string) {
3   },
4   get: function() {
5   }
6 })
```

Listing 6.7: Defining property on run time

Defining such a property can be done on a class instance, making it exclusive to that specific instance, or on a class constructor, making all instances of this class have the newly defined property. A problem arises when combining this property definition with the decorator. Property decorators move the property definition from the class constructor to the class prototype, the differences between the class constructor and class prototype are out of scope for this research. The most important difference to be aware of is that properties that are set on a class prototype do not propagate to a class instance while properties set on a class constructor do propagate properly. This shows that there is a third place to define a property, on the class prototype, this is not often done due to the limitations that are outlined above.

These limitations can be bypassed with a more nested property definition in which you define a property when setting the setter for another function definition. Such a definition can be found in listing A.1. These definitions are hard to read, error-prone, and inefficient in execution. These decorators are easy to use for a developer that wishes to implement this software system into their application, however, from a development standpoint they are terrible to maintain.

6.3.3 Lexical scoping

The last major problem with these decorators is their lexical scoping. In normal JavaScript execution, any line of code can access the data that is declared before it is called, as long as the data resides in its own or one of its parents' lexical scopes. Code can always access its parent scope but never its child scope. Furthermore, the special `this` property is always constant in a lexical scope and matches the last instance that called the specific lexical scope. Within a class method, this would be the class instance, within a function it would be the parent function.

[7] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty`

At multiple moments during the development of this library, these principles seem to have been violated. Due to unknown circumstances which have not been reproducible, the `this` keyword has referenced multiple entities within the same lexical scope. We suspect that comes from an implementation error in the decorator definition.

6.4 Serialization

When the data moves from the server to the browser, it needs to be encoded in some data interchange format, a format which both parties in the communication can understand. Common formats include JSON and XML. JSON or the "JavaScript Object Notation" is the format that RiskChallenger chose to use for its applications. It has a clearly defined grammar and defines strict character encoding schemes which it uses to ensure maximum compatibility [38], the latter of which we already touched on in section 6.2.

To convert any JavaScript variable to the JSON format, the `JSON.stringify` function is used, this will turn the variable into a string representation conforming to the JSON standard. The inverse of this is done by the `JSON.parse` function which turns any JSON string into the proper JavaScript format.

"JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays)." [38] The four primitive types as well as the array structured type have strongly defined serialization in JavaScript which cannot be overwritten. JavaScript `Objects`, however, can overwrite their serialization strategy. This can be used to remove certain properties from an object before serialization or for changing the object structure before serialization. Objects can overwrite their `toJSON` method to overwrite this serialization behavior.

In the context of this software system, we'd like to remove all decrypted properties from the object before sending it to a server. Section 6.3 shows how decorators are used to add new properties to objects which are used to store the decrypted values for easier and faster access. While this way of storing the decrypted values makes applications run faster, since they require decrypting a value only once instead of on each `get` call, it exposes the decrypted value during serialization. This is undesired behavior since the server should never learn any of these unencrypted values. We must therefore ensure that all unencrypted values are removed from the object during serialization.

We can remove these unencrypted values by overwriting the `toJSON` function of any object which has encrypted values. In this function, we can remove the references to the unencrypted values and thus make them safe for transmission to the server. There is however one problem with this solution, the `toJSON` function can not be overwritten using the `Object.defineProperty` call (as we have seen in section 6.3.2). This is due to the `configurable` flag which is set to `false` on the `toJSON` definition, making it impossible to overwrite the `toJSON` function in a decorator.

This problem can be bypassed by the developer of an application that wishes to use our software system. They can overwrite the `toJSON` definition to remove all unencrypted property values. This, however, is very error-prone. It requires modification of all objects that contain encrypted values. Missing a single one of these properties leads to leakage of unencrypted data. A solution that can solely be implemented in our software system has not yet been found.

Chapter 7

JavaScript Crypto Benchmark

Chapter 4 shows a theoretical solution to the design challenge that was laid out by RiskChallenger, to develop a software solution that introduces end-to-end encryption in a plug-and-play type fashion. One of the key elements of any encryption software solution is the encryption algorithm implementation it uses. As previously concluded in section 2.1.1, it is in general not a good idea to implement a cryptography algorithm, therefore requiring the use of an encryption library. The following section includes an experiment in which we compare the runtime speed of three crypto libraries for JavaScript applications. The data collected from this experiment can tell us about the feasibility of using these specific libraries, as well as give us a rough estimate of the overhead that can be expected by introducing end-to-end encryption into a web application. Previous research by the WebKit Team shows that the SubtleCrypto library should be several orders faster than any competing library[31].

The three tested libraries are as follows:

SubtleCrypto
: SubtleCrypto[1] is the built-in JavaScript encryption library that comes included with a browser or more specifically, it is a standard library in the JavaScript engine which comes bundled with a browser. It is an implementation of the WebCrypto standard[32]. Its tight integration with the browser is the reason it is included in this comparison.

Stanford Javascript Crypto Library (SJCL)
: SJCL[2] is a crypto library created by researchers from Standford University. It originally aimed to be a secure and fast library for symmetric encryption [39]. It later evolved to include other common cryptography operations such as key derivation and public/private key cryptography. It is included in this comparison for its scientific and trustworthy background.

Forge
: Forge[3] is a general-purpose crypto library created by Digital Bazaar. It has a plethora of functions and is one of the most actively maintained JavaScript crypto libraries. Its large community of users leads to its inclusion in this comparison.

7.1 Method

The benchmark is performed against three different operations, encryption, decryption, and key derivation. The tests are designed to represent the usage in this library, which means that all encrypted data and the keys are passed into the tests in their encoded format with a string type. Timings, therefore, include the string-to-binary and binary-to-string operations wherever necessary. The code which was executed during these tests can be found in appendix B. All tests were run on the latest version of the relevant operating system, browser, and library as of 26th October 2021.

Each operation is run in 100 sets of 100 executions for a total of 10.000 timed executions. This results

[1] `https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto`
[2] `https://crypto.stanford.edu/sjcl/`
[3] `https://github.com/digitalbazaar/forge`

in 100 data points per operation, each of which indicates how long it took to run 100 executions. Grouping in sets of 100 executions is necessary to gain accurate results since timings lower than 1ms are not measurable to prevent timing attacks[4], this is an extra security feature built-in by most popular browsers.

7.1.1 Key Derivation

The interface to which the key derivation tests needed to conform can be found in listing 7.1, the actual code which was run to test can be found in appendix B.1. Key derivation is done using `PBKDF2`, with 5000 iterations, and yields a 256-bit key. All input data was constant across all test runs.

```
type keyDerivation = (username: string, secret: string) => Promise<string>
```

Listing 7.1: Key derivation test interface

7.1.2 Encryption

The interface to which the encryption tests needed to conform can be found in listing 7.2, the actual code which was run to test can be found in appendix B.2. Encryption is done using `AES-GCM` which is an authenticated encryption system as explained in section 6.1. The plaintext was constant across all encryption tests, the key was constant for all executions of one crypto library but it was regenerated for different crypto libraries. Prior testing indicated that none of these crypto libraries perform any optimization by caching calculated values which could increase their speed. The plaintext is Lorum-Ipsum placeholder text with a size of 4kb.

```
type encrypt = (key: string, plaintext: string) =>
    Promise<{ct: string, iv: string, tag: string}>
```

Listing 7.2: Encryption test interface

7.1.3 Decryption

The interface to which the decryption tests needed to conform can be found in listing 7.3, the actual code which was run to test can be found in appendix B.3. All input data for the decryption test was generated exactly once per encryption library. The ciphertext, iv, and tag were retrieved from the encryption-test function. SJCL was discarded from this test since we were unable to get this library to validate the authentication tag, resulting in the inability to decrypt any data. While this behavior was present in all test cases across all devices and browsers, it could not be replicated with the example code as provided by SJCL.

```
type decrypt = (key: string, data: {ct: string, iv: string, tag: string}) =>
    Promise<string>
```

Listing 7.3: Decryption test interface

[4] https://developer.mozilla.org/en-US/docs/Web/API/DOMHighResTimeStamp#reduced_time_precision

7.2 Results

The tests as outlined above have been run four times, on three different physical machines, with three different operating systems, two browsers have been tested and two different CPU architectures have been used.

Of the tested operations, the key derivation used to generate the user's symmetric key is by far the slowest, the results of benchmarking this operation can be found in figure 7.1. This time intensiveness of the key derivation function was expected and is a desired property since this reduces the impact of brute-force attacks by limiting the number of key derivations that can be done in a given time. Key generation with `Subtle-Crypto` takes on average 1.882 ms, for `SJCL` this is 19.296 ms and `Forge` takes 60.384 ms per operation. The overall spread in speed is very significant, with the fastest time for 100 key generation operations being 67.9 ms while the slowest 100 took 9462 ms. However, the spread within a library is limited to a factor 3 meaning that the slowest 100 operations took at most 3 times as long as the fastest 100, and the spread within a library on a specific device is under a factor 2. The results of this test for the `Forge` run on macOS in the Safari browser shows a significantly larger spread than the other tests show, the source of this behavior is unknown but consistent, reruns of the same experiment on this browser showed similar results. This specific test run was by far the longest running one, it took a total of 14 minutes, with the runner up taking 10 minutes, this might have resulted in the computer performing background tasks that influenced these results.

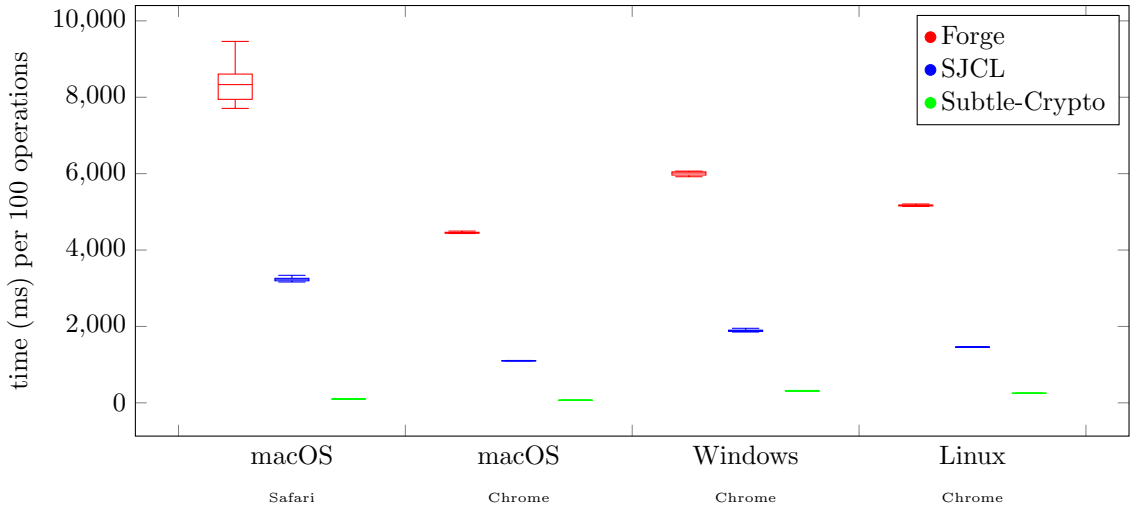


Figure 7.1: Key derivation speeds

Figure 7.2 shows that encryption is significantly faster than key derivation on all devices and for all libraries. We see average single operation speeds of 0.312 ms for `Subtle-Crypto`, `SJCL` scores average speeds of 1.091 ms and `Forge` takes an average of 0.678 ms per operation. Within these results we see significantly more spread on all libraries on the Windows instance, the reason for this is unknown to us, but once more these results were repeatable.

The encryption numbers are similar to the numbers in the case of decryption, the latter of which can be seen in figure 7.3. As discussed in section 7.1.3, this test does not contain any data for `SJCL` since we were unable to acquire the required data due to technical problems. We see average single operation speeds of 0.649 ms in `Subtle-Crypto`, `Forge` is the fastest in decryption with single operation speeds

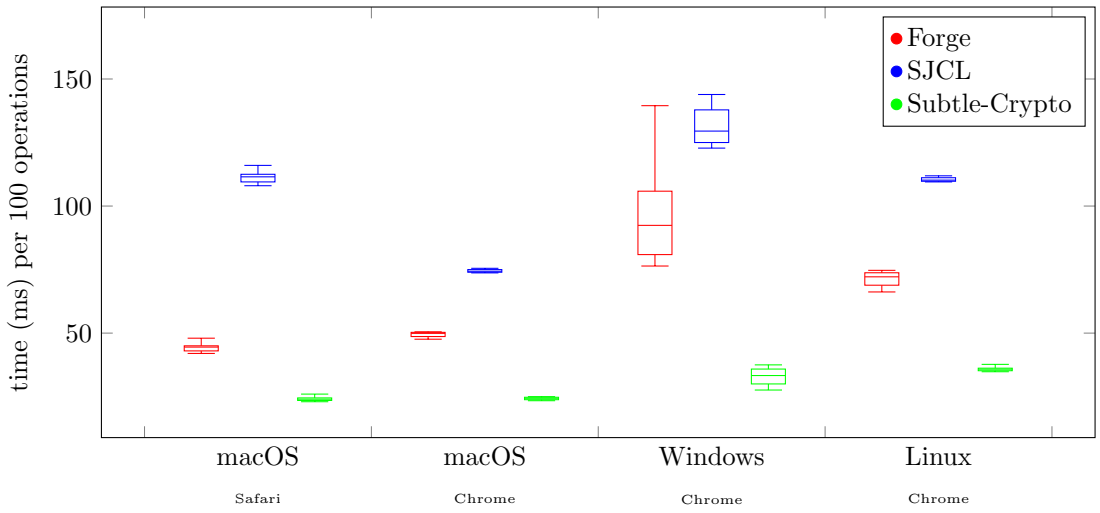


Figure 7.2: Encryption speeds

that average 0.492 ms.

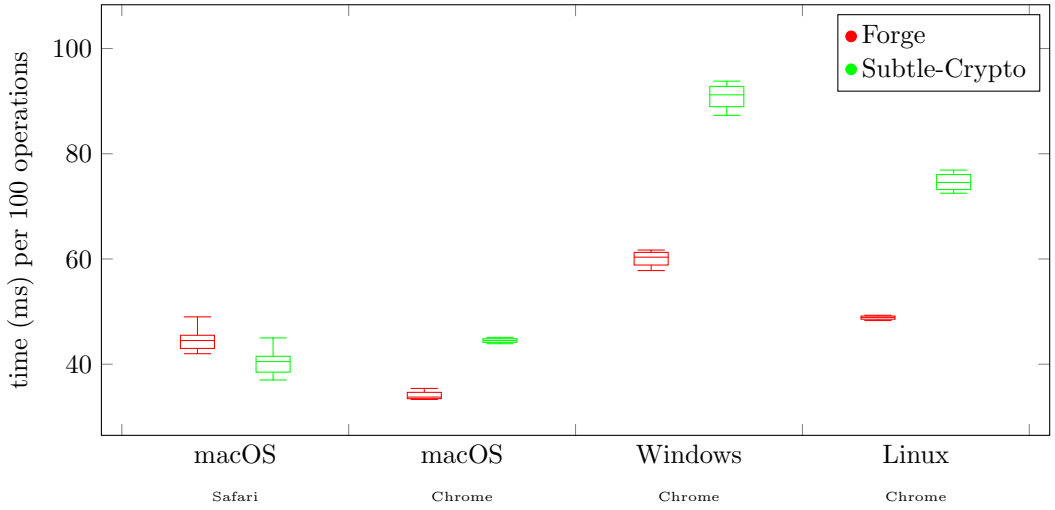


Figure 7.3: Decryption speeds

All of these operations are blazing fast when compared against other operations which are commonly performed in web applications. Loading data from RiskChallenger's server takes hundreds of milliseconds and the default animation transition time is set to 200 milliseconds in RiskChallengers applications. This means that the web application could easily decrypt 200 values before it has opened a modal that can display this information.

Research by the Nielsen Norman Group show that loading times of 0.1 seconds are perceived as instantaneous [40, 41], and while in 2010 load times of 1 second were considered not to break the flow of thought of a user [40], this is no longer the case is 2020 [41]. That leaves a web application with the possibility of decrypting hundreds of values before the addition of our software system will have an impact on the perceived speed of a website.

Given the immense diversity in web applications, it is hard to give an exact number for the amount of overhead that can be expected when implementing our software system. It is however safe to say that

the overhead would be negligible for most web applications. Only the most time sensitive applications such as specialized stock trading applications will need to consider if the added security is worth the overhead.

This benchmark found that speed should not be a predominant factor on which to base the choice of an encryption algorithm, nor should it be a reason not to implement end-to-end encryption. All the benchmarked options perform well enough for most applications.

Chapter 8

Discussion

RiskChallenger asked us to explore the possibilities of developing a software solution that introduces end-to-end encryption in a plug-and-play type fashion. The implementation of such a software system turned out not to be feasible in the given timeframe, we did, however, collect a lot of information that will be useful when implementing said system.

8.1 Results

In chapter 4 we showed a software design of a system that conforms to the wishes and requirements from RiskChallenger. Chapter 5 expands upon this by introducing and providing solutions to some key design principles. Thereafter, chapter 6 explained the problems which were encountered during the implementation of the software for as far as that implementation progressed. Lastly, chapter 7 benchmarked the performance of three crypto libraries to see which of these libraries would be fast enough to be useful in the end-to-end encrypted software solution and to give rough estimates of the overhead that can be expected from implementing this software system.

We believe that these sections give a good foundation on which the desired software system can be built. The system as outlined in chapters 4 to 6 provides security that conforms to the wishes of RiskChallenger as evident from section 4.3. Its aspect-oriented programming based design requires little changes from a developer that wishes to implement the designed system as evident from listings 1.1 and 1.2. It uses many of the design principles of previous research [15–17, 19–23], such as the key usage in which symmetric key encryption is used to encrypt data while asymmetric key encryption is used to encrypt the symmetric keys. Or the use of an identity provider as a trusted third party to distribute public keys, with the possibility of verifying their integrity through fingerprint checking. Where all of the systems as designed by previous research need the software to be built around their system, we designed a system that is easy to implement as an add-on. Security is often still only an afterthought in software development [42]. A software system that can be added as an afterthought to web applications to increase their security is, therefore, welcome for many applications.

Section 7.2 shows that the `SubtleCrypto` library is no longer significantly faster, as it used to be some years ago [31]. Showing that the speed of these crypto-libraries is not a predominant factor in deciding which library to use. It is more important to choose a trustworthy library that fits your requirement. It is difficult to advise on the trustworthiness of each software library, this depends heavily on the specific application that wishes to include our software, their parent company, and their most relevant threats. Choosing the `SubtleCrypto` library seems to be a good choice for most applications since it does not increase the number of people/organizations you need to trust for your security. A bad actor that can compromise the security of the `SubtleCrypto` implementation in a browser could conceivably also modify other parts of the browser and thus leak the data through other means. These results also show that third-party libraries with more features can now be conceivably used, enabling a large new set of functionalities that were previously prohibitively extensive in terms of runtime speed.

8.2 Current Limitations

The work as shown in this research is far from infallible. Especially when compared to the work as shown in chapter 3 we can see reduced security. It is, however, important to consider the difference between the goals of the research from the related work and this research. We focused on creating a software system that is secure yet "usable", a system which will see adaption in the industry - something which has not been achieved by the work from chapter 3. While most research in chapter 3 aims to provide the strongest possible security, the usefulness of this security is void if it is not actually used in real-world systems. We aim to make a lot of websites more secure instead of making a few websites perfectly secure.

8.2.1 Web application integrity

While the system as a whole does protect against a passive attacker who only observes data but does not alter it as shown in section 4.3.3, it fails to protect against active attackers, examples for both of which can be found in section 2.4. Since our design is missing functionality for verification of the integrity of the web application, there are no guarantees that the code which is stored on a web server, is equal to the code being executed by the users' browsers. Mylar [19] has shown that it is possible to give these guarantees. Mylar includes a browser application that cryptographically verifies the integrity of the web pages. This prevents the active attackers as defined in section 2.4 from altering the page by injecting code that might leak confidential information.

Section 2.1.1 already discusses this problem and possible solutions. TLS can be used to guarantee the integrity of the webpage while in transit if both the server and browser can be trusted. However, an active attacker might still be able to alter the webpage on the server before it is sent to the client. The design as outlined by Mylar could be implemented in our software system to protect against this threat.

8.2.2 Trust

As discussed in section 2.1.1, it is important to be aware of who you either explicitly or implicitly trust. The blog "I'm harvesting credit card numbers and passwords from your site. Here's how." by David Gilbertson [43] perfectly illustrates how difficult it is for JavaScript applications to secure themselves. When a developer includes any package, they are trusting the maintainer of that package, is anything malicious present in their source code, is the code they packaged and distribute a compiled version of the public source code? And even if they trust the maintainer of their favorite package, do they trust the maintainers of the packages that their favorite package uses? If an attacker can execute code on a website, the security of that website is pretty much out of the window.

So unless a developer handcrafts all their JavaScript and does not depend on any package, raising the question of why they would include the software system as designed in this research, it is very hard for them to be able to trust the security of their web application.

That does not mean that providing end-to-end encryption in a web application is useless. It will still protect against database dumps, the curious but honest database administrator, and other unauthorized access to data.

8.2.3 Metadata leakage

While protecting metadata is explicitly considered out of scope for this research as stated in section 2.1.4, it is good to acknowledge this fact once more. In certain contexts, leakage of metadata might reveal a lot of information. In the context of this software system, the most prominent metadata is relations between data. The software system as outlined in chapter 4 does not hide what users are members of groups, nor does it hide what groups have access to certain data. It is highly dependant on context if this is acceptable. Knowing how many data entities belong to a certain group, for example by counting the number of related identification numbers of the data, might not be interested in most groups. However, if you know that there exists some data entity with a specific identification number that has information that is interesting for you, you might be able to find out what groups, and thus what users, have access to this data. This might very well help an attacker extract the data through other means such as social engineering.

Collecting metadata is also a common tactic for state agencies since it often requires less burden of proof on the side of the prosecutor as evident from the 2014 case against Ross Ulbricht, creator of the darknet market website Silk Road, in which the FBI state that they "merely collected metadata rather than the actual content of his communications, and thus didn't require proving probable cause to a judge" [44]. It was exactly this metadata that lead to the apprehension of Ross Ulbricht [45], the FBI managed to extract his IP address from metadata, leading them to his location. This clearly shows that ignoring the confidentiality of metadata, as done by this research, is not always the correct choice.

8.2.4 Limited feature set

The design as presented in this research only discusses 3 cryptographic operations. It considers key derivation and both encryption and decryption with both symmetric and asymmetric keys. It does not allow for querying over the encrypted data, searching through encrypted data, or performing calculations over encrypted data. Some of these features are present in the related work.

While these features provide useful security benefits as evident from their uses in the related work, their underlying cryptographical principles are not yet commonly implemented in JavaScript. None of the crypto libraries as tested in chapter 7 have the necessary cryptography implemented to provide these features. This prevents us from implementing these functionalities in our software system.

Furthermore, the features that they provide are not required for all types of applications. For example, most other features that are implemented by CryptDB[15] enable secure computation on a server, a feature which can often but not always be replaced by client-side calculations. Mylar [19] has functionality that allows for server-side full-text search over encrypted data, this is a very powerful feature but the client-side search could replace this functionality in some cases.

8.3 Future Work

This research has the potential to inspire further research projects. We believe that a working implementation of the envisioned software system would still benefit the web development industry, its

developers, and especially users of web applications. Future work could create this working implementation and extend on this or extend upon the ideas as outlined in our research, this section will introduce some of the ideas on which we would like to see further development.

8.3.1 Migration

An interesting question that has not been answered by this research nor by any of the related work has to do with migrating data. How to migrate existing data to an end-to-end encrypted system? It seems logical that this requires encryption of present data but it could well be that there is no key management infrastructure in place, meaning that none of the users have public/private keypairs that can be used to encrypt anything. Developing a strategy on how to migrate an existing database to an end-to-end encrypted database could provide a solution to this problem. Such a strategy could outline the steps required for an application to move towards an end-to-end encrypted system. Likewise, it might be interesting to see how a reverse of this strategy would work, although, in practice, this is rarely seen.

8.3.2 Expanding feature set

Section 8.2.4 mentions the limitations of this research in terms of the features that are considered for implementation. Future research could focus on implementing more of the functionality which is present in `SubtleCrypto` into our software solution. Such as initializing symmetric encryption with a fixed IV to create a deterministic encryption scheme to allow for equality checking or using different key-sizes and block-sizes to allow for more flexibility when implementing the software system in a different application.

It might also be worth exploring feature sets from other crypto libraries, section 8.2.4 already outlines some features which are present in other research and which might be interesting. Especially querying over encrypted data might be interesting in certain applications. Take a movie rating platform as an example, it might be interesting to find all movies with an average review score of 8 or higher while keeping the exact scores hidden. Some applications might also benefit from full-text search, specifically applications that handle a large volume of data for which client-side search is not feasible.

8.3.3 Improved compatibility

During this research, we encountered compatibility-related problems, such as not being able to decrypt data with the forge library which was encrypted using the `SubtleCrypto` library or the other way around. This was mainly due to implementation details of the libraries such as how to pad the blocks to get the correct size or where to store the tag used to validate the encrypted message. These problems could be solved by observing the behavior of the libraries and making small changes in how data was provided to them. We assume that these problems would be harder to solve when working with this data outside of the JavaScript ecosystem, for example when trying to use the data in a native mobile application.

These problems are not just limited to encryption and decryption. Also, the keys have these problems, there exist well-defined key formats and the `SubtleCrypto` system has functions defined to wrap

these keys into portable and compatible formats. It might be worth investigating the uses of these key formats, do they provide better compatibility, and if so at what cost, what is the overhead of wrapping these keys.

8.3.4 Web Authentication API

"The Web Authentication API (also referred to as WebAuthn) uses asymmetric (public-key) cryptography instead of passwords or SMS texts for registering, authenticating, and second-factor authentication with websites." As summarized by Mozilla [46].

The WebAuthn[1] framework is a proposal, awaiting approval to join the JavaScript standard library [47]. It allows for the creation of cryptographic keys which are strongly tied to a domain, preventing phishing attacks. In our research, they can be used as a replacement for the keys belonging to a user. It would no longer require the server to store an encrypted version of the user's private key since these private keys will be stored on the users' device, further reducing the chances of this key getting leaked.

The keys generated by this WebAuthn framework are often backed by hardware security modules that are protected by biometrics. This could lead to increased security as well as increased user experience, making them a great extension to our designed software system.

[1] `https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API`

Chapter 9

Conclusion

This research explored the possibilities of developing a software solution that introduces end-to-end encryption in a plug-and-play type fashion.

We developed a software system that matches the requirements for this software solution. This system is designed to work in a multi-user environment as evident from its special considerations for nested authentication, key distribution, and access revocation. We showed how any directed graph can be used as an access control system allowing for great flexibility. We also discussed common methods of key distribution, their flaws such as the possibility for a man-in-the-middle attack, and ways to improve this process by signing these keys or verifying their fingerprints. Furthermore, we explained ways to revoke access for a user in which we differentiate between a safe-but-efficient method, in which the removed user keeps access to current data but we prevent access to future data by creating and using a new key, and an even more secure option, in which we create a new key and encrypt all current and future data with this new key. Lastly, we outlined threat actors and their common threats to web applications, and we explained how our system can help protect against these threats or where the shortcomings of our system are with regards to these threats. This shows us that our system is secure against a passive attacker, who only observers but does not alter anything, but not against an active attacker who actively alters data. Providing security against an active attacker can be explored in future work.

We worked on implementing the previously mentioned design. This turned out not to be feasible in the given timeframe, however, these attempts did give us useful insights into problems that are expected when implementing the software system. These problems, being a limited set of available encryption algorithms in our crypto-library of choice, difficulties with text-encoding, limitations of decorators, and the inability to overwrite a serialization method, have been documented together with solutions, attempts at a solution, or ways to bypass the problem.

The last contribution of this research is a benchmark that compared the speed of three different JavaScript crypto-libraries, one of which comes bundled with the browser, the second of which comes from a scientific and trustworthy source, and the last of which has a large community and is common in the ecosystem. This benchmark shows that all three of these libraries are so fast that they have a negligible impact on the expected response times of a web application. We show that other common operations performed in web applications take multiple orders of magnitude more time and that hundreds of cryptographic operations can be performed without negatively impacting the user experience.

The parts of this research come together to show that introducing end-to-end encryption is theoretically possible and that introducing end-to-end encryption will not have any significant impact on the speed at which your web application operates. The JavaScript features that can enable such a software solution to work in a “plug-and-play type fashion” are still very new and present challenges for implementation.

Bibliography

[1] Mr. Graham, Mr. Cotton and Mrs. Blackburn. 'S.4051 - Lawful Access to Encrypted Data Act'. In: (15th January 2016), p. 52. URL: https://www.congress.gov/bill/116th-congress/senate-bill/4051/.

[2] Brian Acton and Jan Koum. *end-to-end encryption*. WhatsApp.com. 4th May 2016. URL: https://blog.whatsapp.com/end-to-end-encryption/?lang=en (visited on 08/11/2021).

[3] Dropbox. *Boxcryptor*. Boxcryptor - Dropbox. 2017. URL: https://www.dropbox.com/app-integrations/boxcryptor (visited on 08/11/2021).

[4] Max Krohn. *Zoom Rolling Out End-to-End Encryption Offering*. Zoom Blog. 14th October 2020. URL: https://blog.zoom.us/zoom-rolling-out-end-to-end-encryption-offering/ (visited on 08/11/2021).

[5] Jospeh J. Simons et al. *1923167 - Federal Trade Commission*. 11th September 2020. URL: https://www.ftc.gov/system/files/documents/cases/1923167zoomcomplaint.pdf (visited on 08/11/2021).

[6] J.H. Saltzer and M.D. Schroeder. 'The protection of information in computer systems'. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308. ISSN: 0018-9219. DOI: 10.1109/PROC.1975.9939[1]. URL: http://ieeexplore.ieee.org/document/1451869/ (visited on 11/05/2021).

[7] Jordan Robertson and Michael Riley. 'China Used a Tiny Chip in a Hack That Infiltrated U.S. Companies'. In: *Bloomberg.com* (4th October 2018). URL: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies (visited on 31/10/2021).

[8] Jordan Robertson and Michael Riley. 'The Long Hack: How China Exploited a U.S. Tech Supplier'. In: *Bloomberg.com* (12th February 2021). URL: https://www.bloomberg.com/features/2021-supermicro/ (visited on 31/10/2021).

[9] GlobalStats. *Statcounter Global Stats - Browser, OS, Search Engine including Mobile Usage Share*. StatCounter Global Stats. September 2021. URL: https://gs.statcounter.com/ (visited on 02/11/2021).

[10] Bruce Schneier. *Crypto-gram: October 15, 1998 - Schneier on Security*. 15th October 1998. URL: https://www.schneier.com/crypto-gram/archives/1998/1015.html (visited on 02/11/2021).

[11] MDN Contributors. *Subresource Integrity - Web security — MDN*. 14th October 2021. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity (visited on 02/11/2021).

[12] Julius Ceasar. *Commentarii de bello Gallico*. 1476.

[13] *The three states of information*. The University of Edinburgh. 8th May 2016. URL: https://www.ed.ac.uk/arts-humanities-soc-sci/about-us/information-security-and-governance/what-information-do-i-have-to-protect/the-three-states-of-information (visited on 11/05/2021).

[14] Google. *HTTPS encryption on the web – Google Transparency Report*. 5th May 2021. URL: https://transparencyreport.google.com/https/overview?hl=en (visited on 05/05/2021).

[1] https://doi.org/10.1109/PROC.1975.9939

[15] Raluca Ada Popa et al. 'CryptDB: protecting confidentiality with encrypted query processing'. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP '11*. the Twenty-Third ACM Symposium. Cascais, Portugal: ACM Press, 2011, p. 85. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043566[2]. URL: http://dl.acm.org/citation.cfm?doid=2043556.2043566 (visited on 16/04/2021).

[16] Raluca Ada Popa et al. 'Building web applications on top of encrypted data using Mylar'. In: (29th August 2016), p. 18. URL: https://dl.acm.org/doi/10.5555/2616448.2616464.

[17] Raluca Ada Popa and Nickolai Zeldovich. 'Multi-Key Searchable Encryption'. In: 2013. URL: http://eprint.iacr.org/2013/508 (visited on 23/05/2021).

[18] Paul Grubbs et al. 'Breaking Web Applications Built On Top of Encrypted Data'. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria: ACM, 24th October 2016, pp. 1353–1364. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978351[3]. URL: https://dl.acm.org/doi/10.1145/2976749.2978351 (visited on 23/05/2021).

[19] Raluca Ada Popa. *Mylar*. 1st November 2016. URL: https://css.csail.mit.edu/mylar/security.html (visited on 25/05/2021).

[20] Nikolaos Karapanos et al. 'Verena: End-to-End Integrity Protection for Web Applications'. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016 IEEE Symposium on Security and Privacy (SP). San Jose, CA: IEEE, May 2016, pp. 895–913. ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.58[4]. URL: http://ieeexplore.ieee.org/document/7546541/ (visited on 23/05/2021).

[21] Rishabh Poddar, Tobias Boelter and Raluca Ada Popa. 'Arx: an encrypted database using semantically secure encryption'. In: *Proceedings of the VLDB Endowment* 12.11 (July 2019), pp. 1664–1678. ISSN: 2150-8097. DOI: 10.14778/3342263.3342641[5]. URL: https://dl.acm.org/doi/10.14778/3342263.3342641 (visited on 20/04/2021).

[22] Alexandru Boicea et al. 'Database Encryption Using Asymmetric Keys: A Case Study'. In: *2017 21st International Conference on Control Systems and Computer Science (CSCS)*. 2017 21st International Conference on Control Systems and Computer Science (CSCS). ISSN: 2379-0482. March 2017, pp. 317–323. DOI: 10.1109/CSCS.2017.50[6]. URL: https://ieeexplore.ieee.org/document/7968578.

[23] Fabian Schillinger and Christian Schindelhauer. 'End-to-End Encryption Schemes for Online Social Networks'. In: *Security, Privacy, and Anonymity in Computation, Communication, and Storage*. Ed. by Guojun Wang et al. Vol. 11611. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 133–146. DOI: 10.1007/978-3-030-24907-6_11[7]. URL: http://link.springer.com/10.1007/978-3-030-24907-6_11 (visited on 01/04/2021).

[24] Moxie Marlinspike. *Safety number updates*. Signal Messenger. 17th November 2016. URL: https://signal.org/blog/safety-number-updates/ (visited on 03/11/2021).

[25] Whatsapp. *WhatsApp Encryption Overview - Technical white paper*. 22nd October 2020, p. 13.

[2] https://doi.org/10.1145/2043556.2043566
[3] https://doi.org/10.1145/2976749.2978351
[4] https://doi.org/10.1109/SP.2016.58
[5] https://doi.org/10.14778/3342263.3342641
[6] https://doi.org/10.1109/CSCS.2017.50
[7] https://doi.org/10.1007/978-3-030-24907-6_11

[26] signal. *Signal ¿¿ Documentation*. Signal Messenger. 2013. URL: https://signal.org/docs/ (visited on 07/10/2021).

[27] K. Moriarty, B. Kaliski and A. Rusch. *PKCS #5: Password-Based Cryptography Specification Version 2.1*. RFC8018. RFC Editor, January 2017, RFC8018. DOI: 10.17487/RFC8018[8]. URL: https://www.rfc-editor.org/info/rfc8018 (visited on 03/11/2021).

[28] Ian Goldberg et al. 'Multi-party off-the-record messaging'. In: *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*. the 16th ACM conference. Chicago, Illinois, USA: ACM Press, 2009, p. 358. ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653705[9]. URL: http://portal.acm.org/citation.cfm?doid=1653662.1653705 (visited on 07/10/2021).

[29] Melissa Chase, Trevor Perrin and Greg Zaverucha. *The Signal Private Group System and Anonymous Credentials Supporting Efficient Verifiable Encryption*. 1416. 2019. URL: http://eprint.iacr.org/2019/1416 (visited on 07/10/2021).

[30] Alexis Deveria. *Crypto API: subtle — Can I use... Support tables for HTML5, CSS3, etc.* 10th May 2021. URL: https://caniuse.com/mdn-api_crypto_subtle (visited on 05/10/2021).

[31] Tan Jiewen. *Update on Web Cryptography*. WebKit. Section: Standards. 21st July 2017. URL: https://webkit.org/blog/7790/update-on-web-cryptography/ (visited on 05/10/2021).

[32] Watson Mark. *Web Cryptography API*. 26th January 2017. URL: https://www.w3.org/TR/WebCryptoAPI/ (visited on 05/10/2021).

[33] Kelsey Cairns, Harry Halpin and Graham Steel. 'Security Analysis of the W3C Web Cryptography API'. In: *Security Standardisation Research*. Ed. by Lidong Chen, David McGrew and Chris Mitchell. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 112–140. ISBN: 978-3-319-49100-4. DOI: 10.1007/978-3-319-49100-4_5[10].

[34] Phillip Rogaway. 'Evaluation of Some Blockcipher Modes of Operation'. In: (2nd October 2011), p. 159.

[35] Harry Halpin and Graham Steel. *Security Guidelines for Cryptographic Algorithms in the W3C Web Cryptography API*. Internet Draft draft-irtf-cfrg-webcrypto-algorithms-00. Num Pages: 15. Internet Engineering Task Force, 17th November 2015. URL: https://datatracker.ietf.org/doc/draft-irtf-cfrg-webcrypto-algorithms-00 (visited on 12/10/2021).

[36] Serge Vaudenay. 'Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ...' In: *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT '02. Berlin, Heidelberg: Springer-Verlag, 2nd May 2002, pp. 534–546. ISBN: 978-3-540-43553-2. (Visited on 12/10/2021).

[37] Francesco Berti, Olivier Pereira and Thomas Peters. *Reconsidering Generic Composition: the Tag-then-Encrypt case*. 991. 2018. URL: http://eprint.iacr.org/2018/991 (visited on 12/10/2021).

[38] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments RFC 8259. Num Pages: 16. Internet Engineering Task Force, December 2017. DOI: 10.17487/RFC8259[11]. URL: https://datatracker.ietf.org/doc/rfc8259 (visited on 09/11/2021).

[8] https://doi.org/10.17487/RFC8018
[9] https://doi.org/10.1145/1653662.1653705
[10] https://doi.org/10.1007/978-3-319-49100-4_5
[11] https://doi.org/10.17487/RFC8259

[39] Emily Stark, Michael Hamburg and Dan Boneh. 'Symmetric Cryptography in Javascript'. In: *2009 Annual Computer Security Applications Conference*. 2009 Annual Computer Security Applications Conference (ACSAC). Honolulu, Hawaii, USA: IEEE, December 2009, pp. 373–381. DOI: 10.1109/ACSAC.2009.42[12]. URL: http://ieeexplore.ieee.org/document/5380691/ (visited on 19/10/2021).

[40] Jacob Nielsen. *Website Response Times*. Nielsen Norman Group. 20th June 2010. URL: https://www.nngroup.com/articles/website-response-times/ (visited on 10/11/2021).

[41] Kathryn Whitenton. *The Need for Speed, 23 Years Later*. Nielsen Norman Group. 17th May 2020. URL: https://www.nngroup.com/articles/the-need-for-speed/ (visited on 10/11/2021).

[42] EY. *EY Global Information Security Survey 202*. 18th February 2020. URL: https://assets.ey.com/content/dam/ey-sites/ey-com/en_gl/topics/advisory/ey-global-information-security-survey-2020-single-pages.pdf (visited on 10/11/2021).

[43] David Gilbertson. *I'm harvesting credit card numbers and passwords from your site. Here's how.* HackerNoon.com. 22nd May 2019. URL: https://medium.com/hackernoon/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5 (visited on 20/10/2021).

[44] Andy Greenberg. 'The FBI Finally Says How It 'Legally' Pinpointed Silk Road's Server'. In: *Wired* (5th September 2014). URL: https://www.wired.com/2014/09/the-fbi-finally-says-how-it-legally-pinpointed-silk-roads-server/ (visited on 10/11/2021).

[45] Serrin Turner and Timothy T Howard. 'MEMORANDUM OF LAW IN OPPOSITION TO DEFENDANT'S MOTION TO SUPPRESS EVIDENCE, OBTAIN DISCOVERY AND A BILL OF PARTICULARS, AND STRIKE SURPLUSAGE'. In: (5th September 2014), p. 58.

[46] MDN Contributors. *Web Authentication API - Web APIs — MDN*. 14th September 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API (visited on 04/11/2021).

[47] Jeff Hodges et al. *Web Authentication: An API for accessing Public Key Credentials - Level 3*. 7th October 2021. URL: https://w3c.github.io/webauthn/ (visited on 04/11/2021).

[12] https://doi.org/10.1109/ACSAC.2009.42

Appendix A

Encryption decorator implementation

```
import { findFromKeyLike } from '../crypto/key/find-from-keylike';
import { Key } from '../interfaces';

export const Encrypt = (keyBearingProperty: string) => {
  return function <T extends object, P extends keyof T = keyof T>(
    target: T,
    propertyKey: P
  ) {
    Object.defineProperty(target, propertyKey, {
      set: async function (value: T[P]) {
        const decryptedValues = new WeakMap<T, T[P]>();

        Object.defineProperty(this, propertyKey, {
          get: function (): T[P] {
            return decryptedValues.get(this)!;
          },
          set: async function (value: T[P]) {
            const keyLike: Key = await this[keyBearingProperty];
            const key = await findFromKeyLike(keyLike, {
              value: value,
              target: propertyKey,
              object: this,
            });

            decryptedValues.set(
              this,
              await decrypt<T[P]>(key, value)
            );
            decryptedValues.set(this, value);
          },
          enumerable: true,
        });

        return (this[propertyKey] = value);
      },
    });
  };
};
```

Listing A.1: Encrypt decorator definition

Appendix B

Crypto Libraries Bench-marking setup

The following functions have been called to benchmark the performance of the different crypto libraries.

B.1 Key derivation tests code

```
export async function sjclDeriveKey(
  username: string,
  secret: string
): Promise<string> {
  return sjcl.misc.pbkdf2(secret, username, iterations, keySize).toString();
}
```

Listing B.1: Key derivation test with SJCL

```
export async function forgeDeriveKey(
  username: string,
  secret: string
): Promise<string> {
  return forge.pkcs5.pbkdf2(secret, username, iterations, keySize / 8);
}
```

Listing B.2: Key derivation test with Forge

```
export async function sublteDeriveKey(
  username: string,
  secret: string
): Promise<string> {
  const importedKey = await window.crypto.subtle.importKey(
    'raw',
    fromUtf8(secret),
    {
      name: 'PBKDF2',
    },
    false,
    ['deriveKey', 'deriveBits']
  );
  const derivedKey = await window.crypto.subtle.deriveKey(
    {
      name: 'PBKDF2',
      salt: fromUtf8(username),
      iterations: iterations,
      hash: {
        name: 'SHA-256',
      },
    },
    importedKey,
    {
      name: 'AES-GCM',
      length: keySize,
    },
    true,
    ['encrypt', 'decrypt']
```

```
  );
  return (await window.crypto.subtle.exportKey('raw', derivedKey)).toString();
}
```

Listing B.3: Key derivation test with `SubtleCrpyto`

B.2 Encryption tests code

```
export async function sjclEncrypt(key: string , plaintext: string): Promise<{
  ct: string;
  iv: string;
  tag: string;
}> {
  const response = {};
  const params = {
    mode: 'gcm',
    iv: getRandomBytes(),
  };
  const ctJson: Record<string, unkonwn> = JSON.parse(
    sjcl.encrypt(key, plaintext, params, response)
  );

  return {
    ct: ctJson.ct,
    iv: sjcl.codec.base64.fromBits(response.iv),
    tag: '',
  };
}
```

Listing B.4: Encryption test with SJCL

```
export async function forgeEncrypt(key: string , plaintext: string): Promise<{
  ct: string;
  iv: string;
  tag: string;
}> {
  const buffer = forge.util.createBuffer(plaintext, 'utf8');
  const cipher = forge.cipher.createCipher('AES-GCM', key);
  const ivBytes = getRandomBytes();
  cipher.start({
    iv: ivBytes,
  });
  cipher.update(buffer);
  cipher.finish();
  const encryptedBytes = cipher.output.getBytes();
  const tagBytes = cipher.mode.tag.getBytes();

  return {
    tag: forge.util.encode64(tagBytes),
    iv: forge.util.encode64(ivBytes),
    ct: forge.util.encode64(encryptedBytes),
  };
}
```

Listing B.5: Encryption test with Forge

```
export async function subtleEncrypt(key: string , plaintext: string): Promise<{
  ct: string;
  iv: string;
  tag: string;
}> {
  const ivBytes = getRandomBytes();
  const encrypted = await window.crypto.subtle.encrypt(
    {
      name: 'AES-GCM',
      iv: ivBytes,
    },
    key,
    fromUtf8(plaintext)
  );

  const data = new Uint8Array(encrypted);
  const ct = data.slice(0, data.length - 16);
  const tag = data.slice(data.length - 16, data.length);
  return { iv: toB64(ivBytes), ct: toB64(ct), tag: toB64(tag) };
}
```

Listing B.6: Encryption test with `SubtleCrpyto`

B.3 Decryption tests code

```
export async function forgeDecrypt(key: string, data: {ct: string, iv: string, tag:
    string}): Promise<string> {
  const decIvBytes = forge.util.decode64(data.iv);
  const ctBytes = forge.util.decode64(data.ct);
  const ctBuffer = forge.util.createBuffer(ctBytes);
  const tagBytes = forge.util.decode64(data.tag)
  const tagBuffer = forge.util.createBuffer(tagBytes)

  const decipher = forge.cipher.createDecipher('AES-GCM', key);
  decipher.start({
    iv: decIvBytes,
    tag: tagBuffer
  });
  decipher.update(ctBuffer);
  decipher.finish();

  return decipher.output.toString();
}
```

Listing B.7: Decryption test with Forge

```
export async function subtleDecrypt(key: string, data: {ct: string, iv: string, tag:
    string}): Promise<string> {
  const ivBytes = fromB64(data.iv);
  const dataBytes = fromB64(data.ct);
  const tagBytes = fromB64(data.tag);
  const ctBytes = new Uint8Array([
    ...(new Uint8Array(dataBytes)),
    ...(new Uint8Array(tagBytes))
  ]).buffer

  const decrypted = await window.crypto.subtle.decrypt({
```

```
11      name: 'AES-GCM',
12      iv: ivBytes
13    }, key, ctBytes)
14
15    return toUtf8(decrypted)
16  }
```

Listing B.8: Decryption test with `SubtleCrypto`