Mika Luoma-aho

JavaScript Web Cryptography API

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communications Technology

JavaScript Web Cryptography API

May 18, 2015

Tiivistelmä

Tekijät Otsikko Sivumäärä Aika	Mika Luoma-aho JavaScript Web Cryptography API 56 sivua + 3 liitettä 18.5.2015
Tutkinto	insinööri (AMK)
Koulutusohjelma	tieto- ja viestintätekniikka
Suuntautumisvaihtoehto	ohjelmistotuotanto
Ohjaajat	lehtori Olli Alm

Insinöörityön tavoitteena on analysoida kuinka uusi Web Cryptography API -rajapintamääritys tulee mahdollistamaan tietoturvallisempien selainpohjaisten JavaScript sovellusten toteuttamisen tulevaisuudessa.

Viimeaikaiset kansalaisten tietoturvaan liittyvät paljastukset (mm. Edward Snowdenin paljastama Yhdysvaltain tiedustelupalvelun hyödyntämä laaja valvontakoneisto) sekä viime aikoina tapahtuneet laajamittaiset tietomurrot ovat saaneet kansalaiset huolestuneeksi pilvipalveluihin tallettamansa tiedon tietoturvasta. Tämä asettaa uusia vaatimuksia selainpohjaisen kryptografian ja web-alustan kehittymisen tukemaan uusien kryptografiaa hyödyntävien applikaatioiden tekemistä.

Insinöörityössä tutkittiin, miten uusi Web Cryptography API -rajapintamääritys tulee muuttamaan sitä, kuinka kryptografiaa voidaan toteuttaa selainpohjaisissa sovelluksissa ja kuinka se mahdollistaa uusien kryptografiaa hyödyntävien palveluiden ja sovellusten rakentamisen suoraan JavaScript-ympäristössä.

Työn soveltavassa osassa toteutettiin tietoturvallinen, selainpohjainen sovellus salattujen ja digitaalisesti allekirjoitettujen viestien jakamiseen. Sovelluksessa viestit kulkevat lähettäjältä vastaanottajalle vahvasti salattuna hyödyntäen julkisen avaimen salauksen kryptografiaa, paljastamatta selkokielistä viestiä tai salausavaimia palvelimelle.

Työ paljasti, että Web Cryptography API -spesifikaatio on tärkeä lisäys selainpohjaiselle JavaScriptille ja käytettäessä yhdessä muiden www:n tietoturvaan liittyvien teknologioiden kanssa se mahdollistaa tietoturvallisten selainpohjaisten kryptografiaa hyödyntävien sovelluksien tekemisen. Lisäksi selvisi, että kryptografian oikeanlainen ja tietoturvallinen hyödyntäminen on vaikeaa ja täynnä ongelmakohtia. Ohjelmistokehittäjien tulee perehtyä kryptografiaan liittyvään kirjallisuuteen ymmärtääkseen eri algoritmien ja protokollien vahvuudet ja heikkoudet.

Avainsanat	Kryptografia, JavaScript, WebCryptoAPI, IndexedDB

Author(s) Title Number of Pages Date	Mika Luoma-aho JavaScript Web Cryptography API 56 pages + 3 appendices May 18, 2015
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software Engineering
Instructor(s)	Olli Alm, Senior Lecturer

The purpose of this study is to analyze how recent and upcoming improvements in browser based web cryptography and more specifically the Web Cryptography API help developers to create more secure web applications in the future.

Recently there has been a rise of interest in browser based cryptography, not only because Edward Snowden's revelations about pervasive surveillance by the National Security Agency (NSA) and the United States government interest in the data stored on the cloud services, but also because recent security breaches have made users concerned about their online privacy and data security. All of this has set new requirements for the client-side cryptography and for the web platform standards to evolve to support creation of new (cryptography) applications.

This thesis investigates the challenges and security concerns that developers face when dealing with cryptography on the client-side and the new Web Cryptography API, which brings native cryptographic primitives already implemented in the browser to the JavaScript run-time environment.

As a case study, a proof-of-concept end-to-end secure application for sharing encrypted and digitally signed messages was implemented. The application uses public-key cryptography as a means to encrypt and decrypt messages directly on a web browser, without revealing the plain text message or the master key to the server.

In conclusion, the investigation revealed that the Web Cryptography API specification is an important addition to browser JavaScript capabilities and when combined with other upcoming web security related technologies, it is possible to create cryptographically secure web applications. Finally, it was concluded that creating cryptographically correct applications is hard, full of pitfalls and developers should be proficient with cryptography to be able to understand strengths and weaknesses of algorithms and protocols.

Keywords	Cryptography, JavaScript, WebCryptoAPI, IndexedDB

Acknowledgement

I dedicate this work to my beloved wife Aino whose unconditional encouragement and support made it possible for me to complete this thesis.

I wish to express my heartfelt love to my daughter Frida for coping with the undue paternal deprivation during the course of my study.

To my family, I love you all.

I would like to thank my supervisor, Olli Alm, for the valuable comments and instructions on this work. I would also like to thank my previous employer SC5 Online and current employer Vincit Helsinki Oy, for support and giving me time and flexibility to write this thesis.

This thesis was written with LaTeX document markup language using the “Metropolia Thesis LaTeX Template” (CC BY-SA). Thanks to Panu Leppäniemi and Patrik Luoto for keeping the template up-to-date.

Tuusula, May 18, 2015
Mika Luoma-aho

Contents

Appendices

Abbreviations and Acronyms

AES	Advanced Encryption Standard
API	Application Programming Interface
CORS	Cross-Origin Resource Sharing
DOM	Document Object Model
HMAC	Hash-based Message Authentication Code
HTML5	HyperText Markup Language, Version 5
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IETF	Internet Engineering Task Force
JSON	JavaScript Object Notation
JWK	JSON Web Key
PGP	Pretty Good Privacy
PKI	Public Key Infrastructure
RSA	Rivest-Shamir-Adleman Public-Key Cryptosystem
SHA	Secure Hash Algorithm
SSL	Secure Sockets Layer
TLS	Transport Layer Security
W3C	World Wide Web Consortium
WWW	The World Wide Web
XSS	Cross Site Scripting

1 Introduction

This is an exciting time to be using JavaScript [1], JavaScript having finally outgrown its early reputation as a basic scripting language used to produce simple effects on web pages. When Brendan Eich created the original version of JavaScript for Netscape[1] in 1995, it was meant as a “silly little brother language” for Sun’s[2] Java language [4] which at that time was seen as the only viable way to build portable heavy weight applications [5]. Gartner[3] predicted in 2014 that improved JavaScript performance will begin to push HTML5 [7] and the browser as a mainstream enterprise application development environment [8]. Today JavaScript is used extensively to produce complex rich app-like experiences on browsers and mobile and embedded devices. Projects such as Node.js [9] and IO.js [10] and make it possible to run the same code on the server-side, making JavaScript an even more powerful and more widely adopted language.

Recently there has been a raise of interest in browser based cryptography, not only because recent data breaches [11;12] and fake SSL certificates [13], but also because Edward Snowden’s revelations [14] about pervasive surveillance conducted by the NSA[4] and governments [16]. According to a 2014 survey called “Public Perceptions of Privacy and Security in the Post-Snowden Era” by Pew Research Center [17], 81% of Americans do not feel secure when using social media sites for sharing private information with another person or organization, and 68% feel insecure using chat or instant messages to share private information [18]. Also the latest whitepapers about internet security threats [19;20] from Symantec Corporation showed steep rise on data breaches in 2014, and an ongoing trend for coming years, when people’s personal and financial information get stolen from companies and online services [21].

[1] *Netscape* Communications, an American computer services company, founded in 1994, was a computer services company best known for its Web browser, Navigator. In 1999, Netscape was acquired by AOL. [2]

[2] *Sun* Microsystems, Inc., an American company, founded in 1982, sold computers, computer components, computer software, and information technology services. In 2010, Sun was acquired by Oracle Corporation. [3]

[3] Gartner, Inc. is the world’s leading information technology research and advisory company. [6]

[4] National Security Agency is home to America’s codemakers and codebreakers. The National Security Agency has provided timely information to U.S. decision makers and military leaders for more than half a century [15]

Since an increasing number of web applications are storing confidential data on the web and recent reports show that data breaches of service provider data storages is clearly on the rise [20], there is a desperate need for a client-side cryptography and the web platform standards to evolve to support creation of new type of cryptographically capable web applications. These new applications could utilize cryptography to enable the users to protect their identities and private data, directly on the browser, before the data is transmitted to service provider, mitigating most common data breaches since the service provider then does not have to store passwords or encryption keys that could be used to steal the data.

There is no reason why cryptographic primitives cannot be programmed in plain JavaScript [22], as proven by the **Stanford JavaScript Crypto Library** (SJCL) [23] and other libraries like CryptoJS [24] and Forge [25]. However, these implementations, programmed in plain JavaScript are bound by the limitations of the JavaScript Virtual Machine (such as V8 used by the Google Chrome browser [26]) and the run-time environment they are running in, which have performance problems and cryptography related security gaps that cannot be fixed in plain JavaScript. These include for example missing a cryptographically secure pseudo-random number generator (CSPRNG), missing safe secure-key-storage and non-predictive function execution performance, which makes the current implementations vulnerable to various attacks (see section 2.10 on page 25 for more information on how developers can mitigate these issues) and unsafe to use in real world applications. Furthermore the browser environment is hostile to cryptography [27] and if not correctly secured, it could allow an attacker to compromise the security of the application by injecting malicious code to extract decryption keys or hijack the private data before it is encrypted. Even the authors of the SJCL library say that “Unfortunately, this is not as great as in desktop applications because it is not feasible to completely protect against code injection, malicious servers and side-channel attacks [28]”.

The **Web Cryptography API**[5] specification is a new standard proposal from the Web Cryptography Working Group of the World Wide Web Consortium (W3C) [29] that provides a promising solution to the current situation by giving web developers ability to use the native (low-level machine code) cryptographic primitives already implemented in the web browser, for writing cryptographically secure applications in the JavaScript environment, which was previously only feasible on the server-side environment.

At the time of writing this thesis (May, 2015), the Web Cryptography API specification was in the Candidate Recommendation phase which means that the W3C Working Group has met their requirements satisfactorily for a new standard, and is now waiting for feedback on whether the specification should become a W3C recommendation. Furthermore, for the specification to advance to the next phase, there must be at least two independent, interoperable implementations of each feature [30;31]. Changes to the specification are possible, and developers should check the latest specification, recommendations and usage instructions before starting to use the Web Cryptography API in web applications.

With the introduction of the new Web Cryptography API standard proposal, web browsers are starting to be capable of performing native cryptographic operations, which enables creation of new kind of privacy centered applications and give more options for users to protect their identity and resources online. This thesis aims to investigate why client-side cryptography is needed, although a cryptographically secure channel between the client and the server with HTTP over TLS (HTTPS) [32] already exists and why native implementation of cryptography functions is needed, when cryptography can already be performed using plain JavaScript implementations.

Chapter 2 introduces the Web Cryptography API specification and examines in detail how the specification can be used, what the most common use cases are and what problems it tries to solve. Chapter 3 presents a case study application which uses the new Web Cryptography API specification to enable users to communicate securely without sharing the sent plain text message or the cryptographic keys used to encrypt and decrypt the message with the server. Chapter 4 presents the findings and gives conclusions.

[5] Application Programming Interface (API) defines a set of instructions and standards that developers can use to build applications.

2 Web Cryptography API

2.1 Background

The **Web Cryptography API**[6] is a candidate recommendation specification created by the Web Cryptography Working Group of the World Wide Web Consortium (W3C) [29;33] that describes a cross-platform JavaScript API for performing cryptographic operations in web applications. The specification specifies asynchronous[7] JavaScript API that takes advantage of Web Workers [34] to perform expensive computations, which allows the program flow of the JavaScript application to continue while waiting for the cryptographic operations to be completed.

As the API is meant to be extensible, in order to keep up with future developments within cryptography, the specification does not dictate what algorithms must be implemented by the conforming user agents[8]. Instead it defines a common set of bindings that can be used in an algorithm-independent manner [31].

The API provides a low-level interface for performing cryptographic operations. These operations include encryption and decryption, digital signature generation and verification, hashing and cryptographically secure random number generation. Additionally the API includes operations to generate, manage, import and export key material that can be used with the cryptographic operations.

The API focuses specifically on CryptoKey objects (see section 2.6 on page 21) which provide an abstraction for the underlying raw cryptographic key material. This allows the API to be generic enough to support versatile ways for user agents to expose and store cryptographic key material, without requiring the web application to be aware of the nature of the underlying key storage [31].

[6] Also known as WebCrypto API and Web Crypto API.

[7] In software development, an *asynchronous* operation means that a process operates independently of other processes.

[8] In computing, a user agent is software (for example a web browser) that is acting on behalf of a user.

The use case scenarios for the API includes, but is not limited to, secure messaging, document signing, data integrity protection, cloud storage, multi-factor authentication, protected document exchange, banking transactions, authenticated video services, and encrypted communications via webmail [31;35]. Secure messaging, such as Off-the-Record Message Protocol (OTR) [36], makes it possible to exchange messages between two parties without revealing the encryption keys or other metadata to the server while also protecting previously sent messages properly with forward secrecy [37] by never storing information that is required to decrypt the messages in the future. Cloud storage can be made secure so that the service provider does not have any access to the stored files by allowing the user to encrypt and decrypt files at the client-side before the files are exchanged with the storage service provider.

2.1.1 History

Many JavaScript cryptographic libraries have emerged that give the web applications ability to perform cryptographic operations, such as the **Stanford JavaScript Crypto Library** (SJCL) which is a project by the Stanford Computer Security Lab to build a secure, powerful, fast, small and easy-to-use cross-browser library for cryptography in JavaScript [23]. A whitepaper [38] about SJCL describes the library as an general purpose symmetric crypto-library that is made in JavaScript and that exports a clean interface and is highly optimized. Other libraries are for example Forge [25] and CryptoJS [24] which provide versatile ways for performing cryptographic operations. These are competent libraries, but being plain JavaScript libraries they can be attacked with a variety of different ways as explained in chapter 1 on page 2. Before Web Cryptography API specification, the most promising draft specification in this domain was the DomCrypt API [39] from the Mozilla project. The W3C uses the DomCrypt API as a “straw-man”[9] API.

2.1.2 About Web Cryptography Working Group

The Web Cryptography Working Group (WebCrypto WG) [29] was formed in April 2012 and is part of the Security Activity [41] collaborative effort at W3C. Its charter [42] is to

[9] In software development, straw-man means the initial proposal created to generate discussion and to generate a better proposal [40, 293].

define a high-level API providing common cryptographic functionality to web applications. The building blocks for the charter originates from 2011 when W3C organized an identity in the Browser Workshop to bring active practitioners together to discuss web identity and what can be done to increase security and privacy on the web. The workshop was a great success, with over 80 representatives from various organizations attending the workshop, including participants from the major browser vendors such as Google, Microsoft, Apple, and Mozilla. Also companies such as Netflix, Paypal, and Yahoo! were present. Present was also the security expertise group IETF and the security company RSA [43].

The Web Cryptography Working Group is part of the World Wide Web Consortium (W3C), which is an international community where member organizations, a full-time staff, and the public work together to develop web standards. The mission of W3C is to lead the World Wide Web to its full potential by developing protocols and guidelines that ensure the long-term growth of the Web [33]. W3C has a regional office in Finland which is located at the Tampere University of Technology [44].

2.2 Brief Introduction to Cryptography

Cryptography[10] is used everywhere and is a very important part of daily lives. Cryptography is the science of using mathematics to transform clear text (also called as *plaintext*) to gibberish, also known as *ciphertext* that only the intended recipient with a correct unlock key can transform back to clear text. This process is also known as encrypting (transforming text to ciphertext) and decrypting (transforming ciphertext back to clear text). Cryptography enables secure communication between two or more parties, making banking transactions secure, communicating securely over wireless network (WIFI) hotspots, sending files securely over Bluetooth[11] connection, protecting sensitive documents, protecting content on DVD and Bluray, DRM[12] systems and so on. The following sections present the most common cryptographic building blocks in more detail.

[10] Cryptography comes from the Greek words kryptós, meaning hidden, and graphein, meaning writing; hence cryptography implies hiding the actual message in a written text.

[11] Wireless communication standard used by mobile phones

[12] Digital Rights Management

2.2.1 Public-Key Cryptography

Public-key cryptography or also known as *asymmetric cryptography* is a cryptography system in which a pair of keys, the public and private key, is used to encrypt and decrypt a message respectively. The public and private keys are mathematically related, but it is impossible to derive or deduce the private-key from the public-key [45]. Public-key cryptography is used widely in the data communications and software used today. For example,

- **HTTP Secure (HTTPS)** communications utilize TLS/SSL, which uses public-key cryptography to establish secure communication between two parties.
- **Certificates** provide authenticity of web pages using public-key cryptography to establish trust between a web page and the certificate authority.
- **Secure EMail protocol (S/MIME**[13]**)** and **PGP** (Pretty Good Privacy [46]) uses asymmetric cryptography to protect email messages that are sent between two people.

Given a key pair, data encrypted with the public-key can only be decrypted with its private-key. Conversely, data encrypted with the private-key can only be decrypted with its public-key [47]. This mechanism can be used to exchange data securely, without exposing the decryption key (private-key) to anybody. Public-key can be safely shared with everyone.

Figure 1 shows example how a sender can send an encrypted message to Bob, using Bob's public-key which Bob had shared previously.

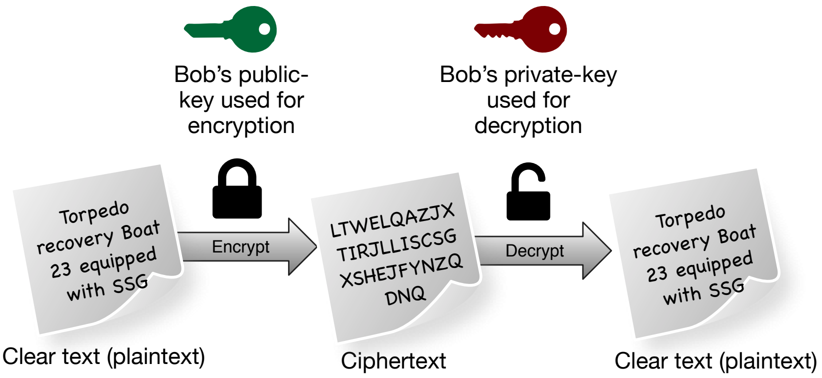

Bob's public-key used for encryption
Bob's private-key used for decryption
Torpedo recovery Boat 23 equipped with SSG
Encrypt
LTWELQAZJX TIRJLLISCSG XSHEJFYNZQ DNQ
Decrypt
Torpedo recovery Boat 23 equipped with SSG
Clear text (plaintext)
Ciphertext
Clear text (plaintext)

Figure 1: Encrypting and decrypting a clear text message with an public and private keys (Copyright © 2015 Mika Luoma-aho)

[13] Secure Multipurpose Internet Mail Extensions

2.2.2 Symmetric Cryptography

Symmetric-key cryptography or also known as *secret-key cryptography* is a cryptography system in which the same key is used for both encrypting and decrypting a message. For sharing the secret-key online, asymmetric cryptography can be utilized. Symmetric cryptography is useful for example,

- **HTTP Secure (HTTPS)** communications utilize TLS/SSL, which uses symmetric cryptography to protect the information exchange.
- **Encrypting large data sets** since symmetric algorithms are fast and can be applied to any data regardless of the data size.
- **Sharing encrypted data** with multiple people without creating a separate key for everyone, since everyone can use the same shared secret-key.
- **Protecting software assets** with a secret-key that is fixed in the software package.

Figure 2 shows how a sender can send an encrypted message using the shared secret key that the sender and recipient have agreed to use.

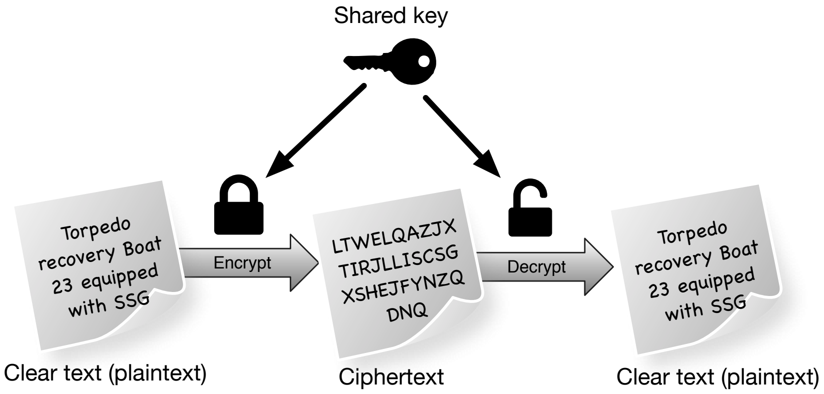
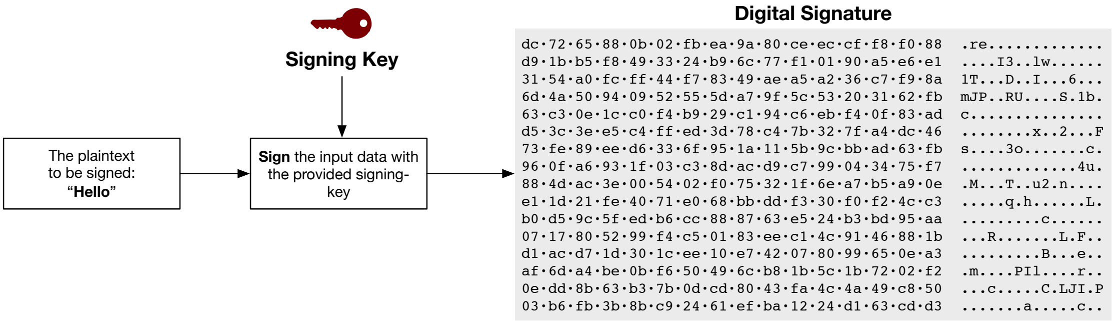

Shared key
Torpedo recovery Boat 23 equipped with SSG
Encrypt
LTWELQAZJX TIRJLLISCSG XSHEJFYNZQ DNQ
Decrypt
Torpedo recovery Boat 23 equipped with SSG
Clear text (plaintext)
Ciphertext
Clear text (plaintext)

Figure 2: Encrypting and decrypting a clear text message with a shared secret (Copyright © 2015 Mika Luoma-aho)

2.2.3 Digital Signature

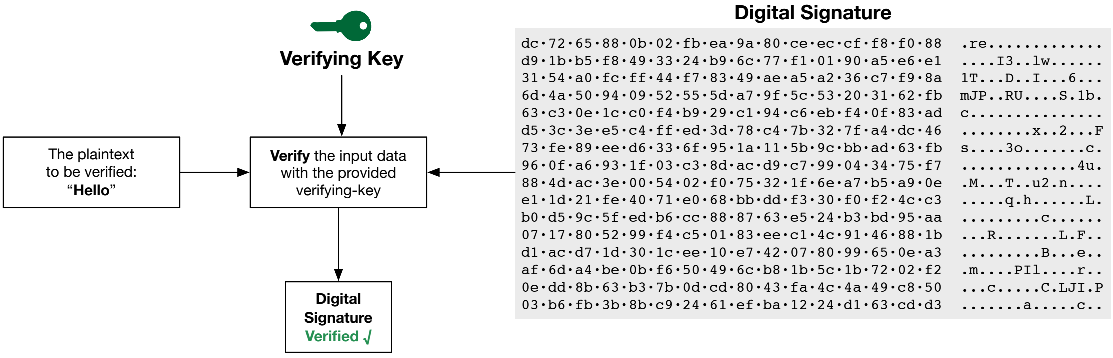
Digital signature is a mathematical scheme, utilizing public-key cryptography for authenticating messages and checking that the document has not been tampered with. A valid digital signature provides a proof, or at least gives the receiver a reason to believe that the message was created by a known sender (providing both authentication and non-repudiation) and that the message has not been tampered with (providing integrity). This is possible because only the sender has the signing-key that could have been used to sign the message. The verifying-key can be attached to the message so that the receiver can easily verify the used key and the digital signature [47]. Figure 3 shows example on how to digitally sign plaintext message.


Signing Key
The plaintext to be signed: "Hello"
Sign the input data with the provided signing-key
Digital Signature

```
dc•72•65•88•0b•02•fb•ea•9a•80•ce•ec•cf•f8•f0•88  .re.............
d9•1b•b5•f8•49•33•24•b9•6c•77•f1•01•90•a5•e6•e1  ....I3..lw......
31•54•a0•fc•ff•44•f7•83•49•ae•a5•a2•36•c7•f9•8a  1T...D..I...6...
6d•4a•50•94•09•52•55•5d•a7•9f•5c•53•20•31•62•fb  mJP..RU....S.1b.
63•c3•0e•1c•c0•f4•b9•29•c1•94•c6•eb•f4•0f•83•ad  c...............
d5•3c•3e•e5•c4•ff•ed•3d•78•c4•7b•32•7f•a4•dc•46  ........x..2...F
73•fe•89•ee•d6•33•6f•95•1a•11•5b•9c•bb•ad•63•fb  s....3o.......c.
96•0f•a6•93•1f•03•c3•8d•ac•d9•c7•99•04•34•75•f7  .............4u.
88•4d•ac•3e•00•54•02•f0•75•32•1f•6e•a7•b5•a9•0e  .M...T..u2.n....
e1•1d•21•fe•40•71•e0•68•bb•dd•f3•30•f0•f2•4c•c3  .....q.h......L.
b0•d5•9c•5f•ed•b6•cc•88•87•63•e5•24•b3•bd•95•aa  .........c......
07•17•80•52•99•f4•c5•01•83•ee•c1•4c•91•46•88•1b  ...R.......L.F..
d1•ac•d7•1d•30•1c•ee•10•e7•42•07•80•99•65•0e•a3  .........B...e..
af•6d•a4•be•0b•f6•50•49•6c•b8•1b•5c•1b•72•02•f2  .m....PIl....r..
0e•dd•8b•63•b3•7b•0d•cd•80•43•fa•4c•4a•49•c8•50  ...c.....C.LJI.P
03•b6•fb•3b•8b•c9•24•61•ef•ba•12•24•d1•63•cd•d3  .......a.....c..
```

Figure 3: Creating digital signature (Copyright © 2015 Mika Luoma-aho)

Figure 4 shows example on how to verify the digital signature.


Verifying Key
The plaintext to be verified: "Hello"
Verify the input data with the provided verifying-key
Digital Signature
Digital Signature Verified √

```
dc•72•65•88•0b•02•fb•ea•9a•80•ce•ec•cf•f8•f0•88  .re.............
d9•1b•b5•f8•49•33•24•b9•6c•77•f1•01•90•a5•e6•e1  ....I3..lw......
31•54•a0•fc•ff•44•f7•83•49•ae•a5•a2•36•c7•f9•8a  1T...D..I...6...
6d•4a•50•94•09•52•55•5d•a7•9f•5c•53•20•31•62•fb  mJP..RU....S.1b.
63•c3•0e•1c•c0•f4•b9•29•c1•94•c6•eb•f4•0f•83•ad  c...............
d5•3c•3e•e5•c4•ff•ed•3d•78•c4•7b•32•7f•a4•dc•46  ........x..2...F
73•fe•89•ee•d6•33•6f•95•1a•11•5b•9c•bb•ad•63•fb  s....3o.......c.
96•0f•a6•93•1f•03•c3•8d•ac•d9•c7•99•04•34•75•f7  .............4u.
88•4d•ac•3e•00•54•02•f0•75•32•1f•6e•a7•b5•a9•0e  .M...T..u2.n....
e1•1d•21•fe•40•71•e0•68•bb•dd•f3•30•f0•f2•4c•c3  .....q.h......L.
b0•d5•9c•5f•ed•b6•cc•88•87•63•e5•24•b3•bd•95•aa  .........c......
07•17•80•52•99•f4•c5•01•83•ee•c1•4c•91•46•88•1b  ...R.......L.F..
d1•ac•d7•1d•30•1c•ee•10•e7•42•07•80•99•65•0e•a3  .........B...e..
af•6d•a4•be•0b•f6•50•49•6c•b8•1b•5c•1b•72•02•f2  .m....PIl....r..
0e•dd•8b•63•b3•7b•0d•cd•80•43•fa•4c•4a•49•c8•50  ...c.....C.LJI.P
03•b6•fb•3b•8b•c9•24•61•ef•ba•12•24•d1•63•cd•d3  .......a.....c..
```

Figure 4: Verifying digital signature (Copyright © 2015 Mika Luoma-aho)

When separate signing and encryption keys are used, the signing key can be revoked if it is lost or compromised, making all future transactions invalid using the same signing key. If the encryption-key is lost, a backup should be utilized. Signing keys should not never be backed up or given to a third party for safe keeping since they could end up in wrong hands.

In several countries, a digital signature has a similar legal status to a traditional signature on a paper. In the European Union, countries have digital signature legislation [48]. Basically this means that anything digitally signed legally binds the signer to the terms and conditions set forth in the signed document.

For reasons set forth above, it is a good idea to use separate key-pairs for signing and encryption. The encryption-key can be used for encrypting normal communication without legally signing every sent message and the signing-key should only be used when both parties have come to an agreement and are ready to be legally binded to a document and to the terms and conditions set forth in the signed document.

It should also be noted that the Web Cryptography API limits the usage of the public and private keys so that it is not possible to generate a key that could be used for both encrypting and signing, so separate signing and verifying keys must be generated (see section 2.10 on page 25 for more information).

2.2.4 Message Digest

A **message digest**, also known as a digest or hash value, is a small unique representation of the plaintext message. A Message digest is created using hashing algorithms (such as MD5 [49], SHA-1, SHA-256, SHA-512 [50]), which are one-way encryption algorithms, meaning that it is impossible to derive the original message from the digest. Given the same input, the output of a hash algorithm is always the same.

Figure 5 on the following page shows example for creating a message digest using an SHA-1 algorithm which outputs 160 bits (20 bytes) of message digest data. The input string is on the left and output on the right side. Output is shown first with hexadecimal display and then in ascii format. Even small change on the input string changes the output drastically.

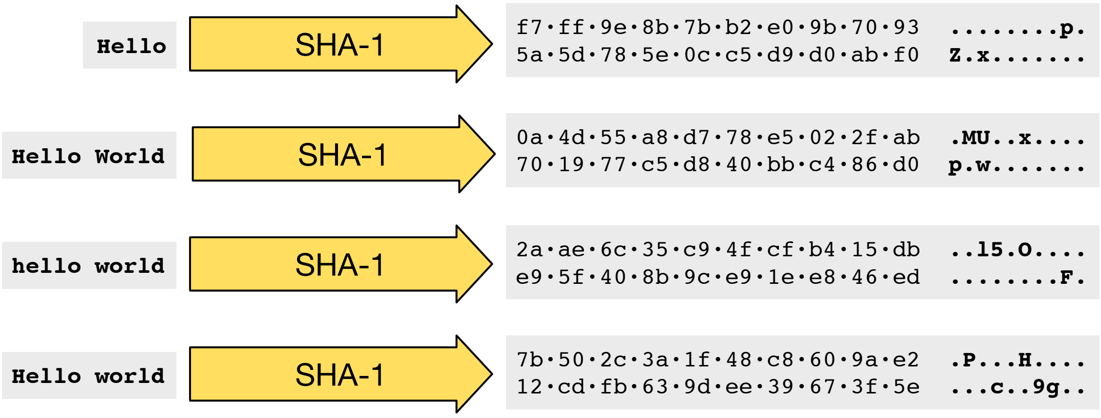

Hello
SHA-1
f7•ff•9e•8b•7b•b2•e0•9b•70•93 ....p.
5a•5d•78•5e•0c•c5•d9•d0•ab•f0 Z.x.......
Hello World
SHA-1
0a•4d•55•a8•d7•78•e5•02•2f•ab .MU..x....
70•19•77•c5•d8•40•bb•c4•86•d0 p.w.......
hello world
SHA-1
2a•ae•6c•35•c9•4f•cf•b4•15•db ..l5.O....
e9•5f•40•8b•9c•e9•1e•e8•46•ed ....F.
Hello world
SHA-1
7b•50•2c•3a•1f•48•c8•60•9a•e2 .P...H....
12•cd•fb•63•9d•ee•39•67•3f•5e ...c..9g..

Figure 5: Creating message digest with SHA-1 algorithm (Copyright © 2015 Mika Luomaaho)

Some of the older hashing algorithms that use a small number of bits to represent the hash can be prone to collisions, where two different messages output exactly the same message digest. To mitigate this vulnerability, modern hashing algorithms using 256 or more bits, such as SHA-256 or SHA-512 (256 and 512bits respectively), should be selected. In fact, old hashing algorithms, such as MD5 (128bits) and SHA-1 (160bits), are usually only provided for backward compatibility but should not be used for creating new message digests.

Message digests are useful for:

- **Checking data integrity**, even small change in data produces a drastically different digest.
- **Creating a unique identifier** for data assets that can be used as a key when storing the data in database.

Additionally a message digest has a small footprint (for SHA-256 algorithm, only 256 bits or 32 bytes [50]) also it is very hard to find two different messages that would produce the same message digest value and hashing algorithms are faster than any encryption algorithm (asymetric or symmetric).

2.3 Basic Usage of the Web Cryptography API

The **Crypto** interface [51] represents the main interface of the Web Cryptography API for performing general purpose cryptographic functionality. The interface is exposed to window [52] and worker [53] objects as a **crypto** object (in the web browser run-time environment) which means that the *crypto* object is globally accessible from JavaScript. The *crypto* object contains methods for creating cryptographically secure pseudo-random numbers and a **subtle** object which implements the **SubtleCrypto** [54] interface.

The **SubtleCrypto** interface provides access to common cryptographic primitives, such as hashing, signing, encryption and decryption. The “SubtleCrypto” name reflects the fact that many of supported algorithms have subtle usage requirements in order to provide the required algorithmic security guarantees [31].

When browser vendors are in middle of implementing a new feature, such as Web Cryptography API, which is not yet considered a common standard, a special vendor prefixing is usually used. The format follows “vendor identifier” + “Name of the function or property” scheme. Vendor identifiers are “ms” for Microsoft Explorer, “moz” for Mozilla Firefox and “webkit” for WebKit based browsers such as Safari and Chrome. However in case of Web Cryptography API, most of the browsers already expose the interfaces without vendor prefixes, but Safari support is still experimental and thus exposes the *SubtleCrypto* interface using a vendor prefixed *webkitSubtle* object. However most of the functionality is already supported in Safari and can be used, but the *SubtleCrypto* interface needs to be remapped so that existing code can be run without modifications. Listing 1 shows how the *SubtleCrypto* interface could be remapped at the start of the main JavaScript application, so that the *SubtleCrypto* interface can be found at standard location (window.crypto.subtle).

```
// fix safari crypto namespace
if (window.crypto && !window.crypto.subtle && window.crypto.
    webkitSubtle) {
  window.crypto.subtle = window.crypto.webkitSubtle;
}
```

Listing 1: Fixing Safari crypto namespace

To detect if Web Cryptography API is available, it is enough to test that *window.crypto* and *window.crypto.subtle* exist. The code in listing 2 defines a function *isWebCryptoAPISupported()* which performs the detection and returns a Boolean value indicating success (true) or failure (false).

```
/**
 * Detect Web Cryptography API
 * @return {Boolean} true, if success
 */
function isWebCryptoAPISupported() {
    return 'crypto' in window && 'subtle' in window.crypto;
}
```

Listing 2: Web Cryptography API feature detection

The **Crypto** interface defines the function **getRandomValues** that lets developers get cryptographically random values. The function is synchronous, meaning that it will block the control of the program flow until it has completed execution. The function takes one argument, the Array, which can be any integer type (for example, Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array, or Uint32Array), which is then filled with random values (random in its cryptographic meaning) [55]. To guarantee enough performance, web browsers are not required to use a truly random number generator, but they can use a pseudo-random number generator [56] (PRNG) seeded with a value with enough entropy. The PRNG used differs from one browser to the other but is suitable for cryptographic usages. Web browsers are also required to use a seed with enough entropy, such as a system-level entropy source [55].

Listing 3 shows JavaScript code that can be used to generate random values.

```
// get 10 random numbers as unsigned 8-bit integers (value range is 0-255)
var size = 10;
var array = new Uint8Array(size);
window.crypto.getRandomValues(array);

// print values to console
for (var i=0; i!==array.length; ++i) {
    console.log(array[i]);
}
```

Listing 3: Generating random values

The **SubtleCrypto** interface defines common methods for performing cryptographic operations. These methods include the following:

- **Encrypt** and **decrypt** methods for transforming plaintext to ciphertext and vise versa using asymmetric[14] and symmetric[15] algorithms. The following algorithms are supported:
  - **RSA-OAEP** [57] is a public-key (asymmetric cryptography) encryption scheme combining the RSA algorithm [58] with the Optimal Asymmetric Encryption Padding (OAEP) method [59].
  - **AES-CTR** [60] is a symmetric-key encryption scheme using AES in the Counter mode.
  - **AES-CBC** [60] is a symmetric-key encryption scheme using AES in the Cipher Block Chaining mode.
  - **AES-GCM** [61] is a symmetric-key encryption scheme using AES in the Galois/Counter mode.
  - **AES-CFB** [60] is a symmetric-key encryption scheme using AES in the Cipher Feedback mode.
- **Sign** and **verify** methods for protecting documents and other assets with a digital signature that can be used to verify authenticity and integrity. The following algorithms are supported:
  - **RSASSA-PKCS1-v1.5** [57] is an RSA Signature Scheme with an appendix based on PKCS #1 v1.5.
  - **RSA-PSS** [57] is a RSA Signature Scheme with an appendix based on a Probabilistic Signature Scheme.
  - **ECDSA** [62] is an Elliptic Curve Digital Signature Algorithm.
  - **AES-CMAC** [63] is a symmetric authentication algorithm based on Cipher-based Message Authentication Code.
  - **HMAC** [64] is a cryptographic hash algorithm.
- **Digest** method for calculating a short digital fingerprint of data that can be used for checking data integrity or used as a unique identifier for assets. The following algorithms are supported:
  - **SHA-1**, **SHA-256**, **SHA-384**, **SHA-512** [50]

[14] In *asymmetric* cryptography, a pair of keys, the *public* and *private*, is used to encrypt and decrypt messages respectively.

[15] In *symmetric* cryptography, a single, common key, is used to encrypt and decrypt messages.

- **GenerateKey** method for generating symmetric and asymmetric **CryptoKey** objects (see section 2.6 on page 21) that can be used with operations that require keys, such as *encrypt* or *sign*.
- **DeriveKey** and **deriveBits** methods for deriving a secret key or bits from a key material or other information such as a password or passphrase using a pseudo-random function.
- **WrapKey** and **unwrapKey** methods for protecting cryptographic key material with another key for example when key material must be either transmitted over insecure communication channels or stored within untrusted environments.
- **ImportKey** and **exportKey** methods for importing and exporting key material from and to different formats.

2.4 Execution Model

All of the methods from *SubtleCrypto* interface return a **Promise** [65]. Promise is a paradigm in JavaScript programming which can be used to replace nested callback function calls (nested callbacks are usually referred to as “Callback Hell” or “Pyramid of Doom” [66;67]) making asynchronous programming more natural and JavaScript code much readable. Promise is an upcoming feature from the next version of JavaScript (ECMAScript 6) [68]. Promises enable the application flow to continue asynchronously while waiting for a promise of results to arrive. A promise can be in any of three states: pending, fulfilled, or rejected. Promise is in a pending state while it is waiting for results and will will trigger fullfilled or rejected state depending on the outcome of the execution. When the promise comes to a fullfilled state (also called a “resolved” state), the promise will call *then()* handler with the results. If the promise comes to a rejected state, the promise will call the *catch()* handler with the information about why the promise was rejected, usually returning the JavaScript Error object [69].

Figure 6 on the following page shows how promises work when calling *SubtleCrypto* interface method or any other method that return a *promise*. The steps are specified below.

1. Method is called
2. Promise is returned in pending state (while the method is executing)
3. If method execution is successful, the promise changes state to fullfilled (resolved)

4. Result from the method execution is returned from the promise
5. If method execution fails for any reason, the promise changes state to rejected
6. Error is returned from the promise

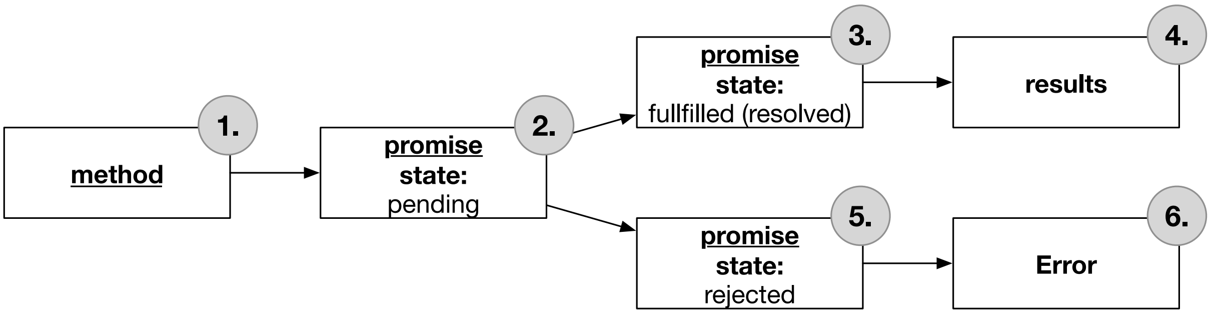


Figure 6: Calling method that return a *promise*

2.5 Example Scenarios

All of the example scenarios presented in this thesis utilize helper module, called utilities module (utils.js) which was written for this project and provide helpful methods for handling input and output data in a various formats that the *SubtleCrypto* interface methods require. For example, in order to convert plain text into Uint8Array (Typed array that represents an array of 8-bit unsigned integers) the utility module provides convertTextToUint8Array method that can be utilized. A Complete copy of the utils.js module can be found in appendix 2 and also in the GitHub repository [70].

2.5.1 Encrypting and Decrypting a Message

One of the most common use cases for cryptography is to transform clear text to ciphertext. The following example application shows how to generate keys needed to perform cryptographic operations and how to encrypt the plain text message to ciphertext and how to decrypt the ciphertext back to plain text using asymmetric (public-key) cryptographic algorithm.

In listing 4 on the next page, the application defines local variables, the message to be encrypted (line 1), temporal storage for generated CryptoKey (key-pair) object (line 2), temporal storage for encryption and decryption results (line 3-4), and the RSA-OAEP [57;

58] encryption scheme with 2048bit key-size as a algorithm to be used for cryptographic operations in this example application (lines 5-12).

```
 1 var message = 'Lorem ipsum dolor sit amet, consectetur
      adipiscing elit.';
 2 var cryptoKeyPair; // for storing the CryptoKey object
 3 var ciphertext; // for storing the encryption result
 4 var plaintext; // for storing the decryption result
 5 var algorithm = {
 6   name: "RSA-OAEP",
 7   modulusLength: 2048, // 1024, 2048, 4096
 8   publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
 9   hash: {
10     name: "SHA-256" // SHA-1, SHA-256, SHA-384, SHA-512
11   }
12 };
```

Listing 4: Defining variables

In listing 5, the application generates a new key-pair for encrypting and decrypting using *crypto.subtle.generateKey* method (lines 14-18). If the key generation succeeds the resulting CryptoKey object is stored in cryptoKeyPair object (line 21).

```
13 // Generate keys
14 window.crypto.subtle.generateKey(
15   algorithm,
16   false, // non-exportable
17   ["encrypt", "decrypt"] // usage
18 )
19 .then(function(result) {
20   // Store keys
21   cryptoKeyPair = result;
22 })
```

Listing 5: Generating keys

In listing 6, the application performs the encryption by converting the input text to Uint8Array format (which is required format by subtle.encrypt method) and then encrypting the message using the *crypto.subtle.encrypt* method with the public-key from the CryptoKey object (lines 25-30). If the encrypt method succeeds the resulting ArrayBuffer formatted data is converted to Uint8Array format and then stored in ciphertext variable (line 34).

```
23 .then(function() {
24   // Encrypt
25   var data = utils.convertTextToUint8Array(message);
26   return window.crypto.subtle.encrypt(
27     algorithm,
28     cryptoKeyPair.publicKey,
29     data
30   );
31 })
32 .then(function(result) {
33   // Store ciphertext
34   ciphertext = new Uint8Array(result);
35 })
```

Listing 6: Encrypting a message

In listing 7, the application displays the encrypted data in console. The convertUint8ArrayToHexView helper method from utils module is used to format the ciphertext to hex format (lines 38-40).

```
36 .then(function() {
37   // Output
38   console.log('Encrypted data:');
39   console.log(
40     utils.convertUint8ArrayToHexView(ciphertext, 16, ''));
41 })
```

Listing 7: Displaying the encrypted data in console

In listing 8, the application performs the decryption by using the *crypto.subtle.decrypt* method with the private-key from the CryptoKey object (lines 44-48). If the decrypt method succeeds the resulting ArrayBuffer formatted data is converted to Uint8Array format and stored in plaintext variable (line 52).

```
42 .then(function() {
43   // Decrypt
44   return window.crypto.subtle.decrypt(
45     algorithm,
46     cryptoKeyPair.privateKey,
47     ciphertext
48   );
49 })
50 .then(function(result) {
51   // Store plaintext
52   plaintext = new Uint8Array(result);
53 })
```

Listing 8: Decrypting a message

In listing 9, the application displays the decrypted data in console. The convertUint8ArrayToText helper method from utils module is used to format the Uint8Array data to text format (lines 56-57).

```
54 .then(function() {
55   // Output
56   console.log('Decrypted data:');
57   console.log(utils.convertUint8ArrayToText(plaintext));
58 })
```

Listing 9: Displaying the decrypted data in console

In listing 10, the application defines an error handler which displays error in the console (lines 60-64). The error handler is called if promise is rejected by any of the previous steps.

```
59 .catch(function(err) {
60   console.error(
61     'Error code:', err.code,
62     ', name:', err.name,
63     ', message:', err.message
64   );
65 });
```

Listing 10: Encrypt and decrypt (Catch errors)

Finally, in listing 11, the application outputs the results to the console. First the encrypted data is displayed using hexadecimal format on the left side and plain text is displayed on the right side and then below the decrypted data is displayed as a plain text.

```
Encrypted data:
[length: 256 bytes (2048 bits)]
ad56a26a1b1e09515a60832c724c88b2  .V.j...QZ...rL..
2d950f5b6f741670fe5149d3b683f7ff  ....ot.p.QI.....
0a7ab6a608477cf7e76576ffe44647bb  .z...G...ev..FG.
dfe863e843208f99b900152b415678d6  ..c.C.......AVx.
b96871f0c30cc046dea60d67fd4bbc64  .hq....F...g.K.d
0e09d310b52c85d48bcc7b01b56ea940  .............n..
32e834f65096f274f036af8b36a3caaa  2.4.P..t.6..6...
20638403a90207e05d00b8280bd3d2dd  .c..............
4614f7ee22800c17f62a931613f724d4  F...............
d2f43a62e04f96fc649a98c8f0554f3b  ...b.O..d....UO.
86360b43ff2d20ab8ca61c1c6101c9b6  .6.C........a...
52962e46e6a48fbae6f167fee3115eec  R..F......g.....
a59f8fc8c8fa1fa759a2e386c7a54a85  ........Y.....J.
cbe6fba09ca20cfa77544533fc4c2bdc  ........wTE3.L..
```

```
c5c5283f8d53d2d63e5b08013cead68f  .....S..........
f5aa7495d7580de2e0821b007d6a98e6  ..t..X.......j..

Decrypted data:
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
```

Listing 11: Console output

2.5.2 Exporting Public-Key from CryptoKey Object

In cryptography it is useful to export the public-key or secret-key out of the application for sharing it with other users or applications. The following example shows how to export the public-key in the JSON WebKey (JWK) [71] format.

In listing 12, the application defines an variable for specifying algorithm and the settings to be used (lines 1-8), generates a key-pair using RSA-OAEP algorithm (lines 10-14) and then exports the public-key part of the CryptoKey object (lines 16-19) in JWK format and then finally displays the resulting JSON object in the console (lines 22-23).

```
 1 var algorithm = {
 2   name: "RSA-OAEP",
 3   modulusLength: 2048, // 1024, 2048, 4096
 4   publicExponent: new Uint8Array([0x01, 0x00, 0x01]),
 5   hash: {
 6     name: "SHA-256" // "SHA-1", "SHA-256", "SHA-384", "SHA
          -512"
 7   }
 8 };
 9
10 window.crypto.subtle.generateKey(
11   algorithm,
12   false, // non-exportable
13   ["encrypt", "decrypt"] // usage
14 )
15 .then(function(cryptoKey) {
16   return window.crypto.subtle.exportKey(
17     'jwk', // export format
18     cryptoKey.publicKey
19   );
20 })
21 .then(function(exportedKey) {
22   console.log('Exported key:');
23   console.log(JSON.stringify(exportedKey));
24 });
```

Listing 12: Exporting key

Listing 13 shows the complete output of the application. The output is displayed using stringified JavaScript Object Notation (JSON) format.

```
Exported key:
{
  "alg":"RSA-OAEP-256",
  "e":"AQAB",
  "ext":true,
  "key_ops":["encrypt"],
  "kty":"RSA",
  "n":"rJfyCYI2uVa_IWviXTFljMPgO_iwZVSh-
       ZoYtW9kUXmyAtcpaNBOlcHYVtqVe3wdRS
       LE-SEO6m08QXU7v63d9m0vUoWzqnXaWzN
       fJaP_2CfhcC_k2DWWprJxY6r0gykMsm6X
       QkJldfmOO55CQ4U_vnv55xJUa_AppGFdg
       2x-FsXBNYUb5krEw-TODQHSEcCzk6d_cz
       iBM41WHuea2GROXPMeyi_jAHt-tDEdDBl
       YQD1IW4tqXL7U9XYUCO4JMlSuKlarNmju
       2ygrgWUvpySVYEh5OHSLTknX24GiBS48l
       esc1pLK-NRTsSMA4KXYgnkBbBnkwUdGZH
       HNmyLC1xRrow"
}
```

Listing 13: Exporting key

2.6 CryptoKey Objects

The **CryptoKey** object is used as a reference to the key material that is managed by the user agent. CryptoKey objects may reference key material that has been generated and imported by the user agent or key material that has been derived from other keys by the user agent or made available to the user agent in some other ways. Also the CryptoKey object does not necessary directly interact with the underlying key storage mechanism, and may instead simply instruct the user agent how to obtain the key material when needed, for example when performing a cryptographic operation. CryptoKey object can hold reference to a asymmetric key, key-pair (public-key and/or private-key), or symmetric key (secret-key). The CryptoKey object also contains information about the algorithm and the settings that were used when generating the key(s).

The specification does not explicitly provide any new storage mechanisms for CryptoKey objects. Instead it allows the CryptoKey objects to be used with any existing or future web storage mechanism that supports storing structured clonable objects. It is expected that in practice most developers will make use of the Indexed Database API (IndexedDB) [72] which allows associative storage of key/value pairs, where the key can be used as an identifier for the key object and the value for storing the CryptoKey object.

Indexed database allows storing and retrieving the CryptoKey objects, without ever exposing the object to the application or to the JavaScript environment. Also since the Indexed Database API uses same-origin access-policy, the application can only access the keys that were previously stored by the same origin, thus securing the key usage [31] (for more information about same-origin concept, see the section 2.10 on page 25).

CryptoKey objects can be stored in any storage that supports a structured clone algorithm [7]. Indexed Database API supports the structured clone algorithm and can be used to store CryptoKey objects. LocalStorage only supports storing simple objects and is not suitable for storing CryptoKey objects.

Previously the main Web Cryptography API provided the KeyStorage interface for discovering and storing pre-provisioned cryptographic keys, which are keys that have been made available to the User-Agent by means other than the generation, derivation, import and unwrapping functions of the Web Cryptography API, however a privacy issue of using these kind of keys as “super-cookies” was identified and that caused the removal of the functionality and creation of another new specification, the WebCrypto Key Discovery API (see section 2.9 on page 24).

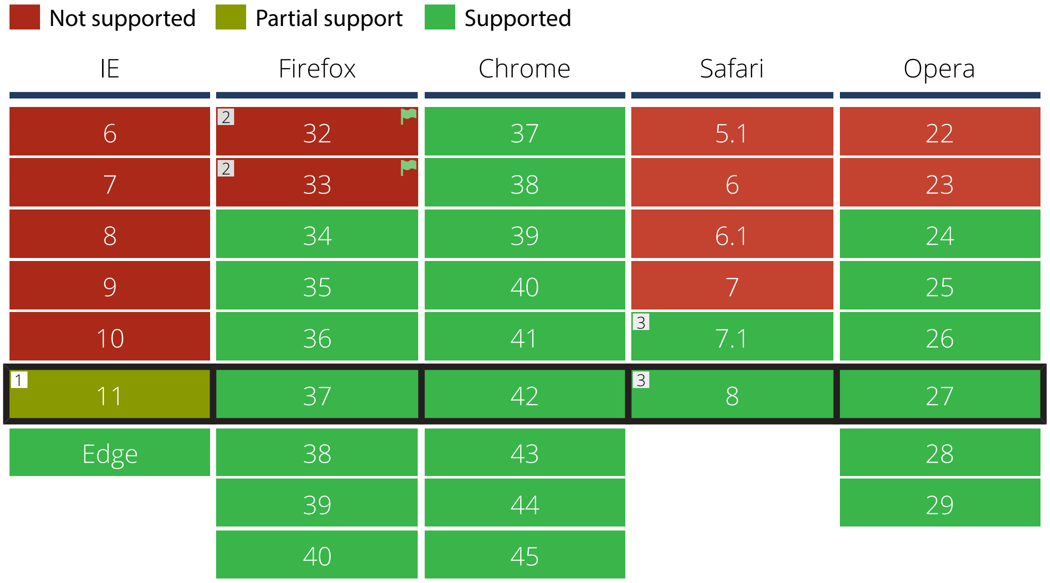
2.7 Browser Support

Now that the specification is in the Candidate Recommendation phase, all of the major browser vendors are making the API available. Since the API does not mandate any particular algorithms, it is possible that different browsers (user agents) support different algorithms and some algorithms may be deprecated once they are deemed insecure. Thus compatibility between different versions of browsers and the API implementation cannot be guaranteed [31].

The latest information about WebCrypto API support can be found on browser vendor support pages and from caniuse.com ("Can I use" web site) which provides up-to-date browser feature support tables for desktop and mobile web browsers [73]. Figure 7 shows the support available in different desktop web browsers (and browser versions) and currently the globally most used browser versions (marked with black border lines).

The figure contains the following notes,

1. Support in IE11 is based an older version of the specification.
2. Supported in Firefox behind the dom.webcrypto.enabled flag.
3. Supported in Safari using the crypto.webkitSubtle prefix.



🟥 Not supported 🟨 Partial support 🟩 Supported

IE	Firefox	Chrome	Safari	Opera
🟥 6	🟥 [2] 32	🟩 37	🟥 5.1	🟥 22
🟥 7	🟥 [2] 33	🟩 38	🟥 6	🟥 23
🟥 8	🟩 34	🟩 39	🟥 6.1	🟩 24
🟥 9	🟩 35	🟩 40	🟥 7	🟩 25
🟥 10	🟩 36	🟩 41	🟩 [3] 7.1	🟩 26
🟨 [1] 11	🟩 37	🟩 42	🟩 [3] 8	🟩 27
🟩 Edge	🟩 38	🟩 43		🟩 28
	🟩 39	🟩 44		🟩 29
	🟩 40	🟩 45		

Figure 7: Browser support (Source: caniuse.com by Alexis Deveria, used under CC BY-NC 3.0 license [73]).

2.8 Polyfills

A polyfill term was first introduced by Remy Sharp [74] and has been since used with the connection of libraries that try to fill in the missing parts from browser implementations [75]. JavaScript Polyfills have existed for a number of years and enable developers to take advantage of upcoming (or in most cases, current) APIs across browsers, old and new by implementing similar behavior than the native implementation. Polyfills can roughly be divided into two categories, extensions to the core Document Object Model (DOM)[16]

[16] The Document Object Model (DOM) is an application programming interface (API) for HTML and XML documents

and Browser Object Model (BOM)[17], or extensions to the core JavaScript language. In the case of Web Cryptography API, the polyfill libraries were useful in the early days of the specification, since there weren't any native implementations available for developers to try out. However, now that most major browser vendors are currently implementing support for Web Cryptography API, the polyfill libraries are not needed anymore and most of them have discontinued development.

The United States Department of Homeland Security and BBN Technologies created **PolyCrypt** [76] polyfill that implemented the 2012 draft specification of the Web Cryptography API which developers could use to get a feel for how they can use the API in practice. PolyCrypt is no longer under active development now that Web Cryptography API is in its final stages to be released.

Netflix created **NfWebCrypto** [77] polyfill as a native browser plugin. NfWebCrypto does not implement the Web Cryptography API in its entirety, due to limitations of browser plugin technology. NfWebCrypto focus is on the operations and algorithms most useful to Netflix. However, the existing feature set supports many typical and common crypto use cases targeted by the Web Cryptography API. The NfWebCrypto library is no longer under active development now that the Web Cryptography API is in its final stages to be released.

2.9 Related Specifications

WebCrypto Key Discovery[18] is an upcoming specification created by the Web Cryptography Working Group of the World Wide Web Consortium (W3C) [29;33], that defines an API that allows discovering named, origin-specific pre-provisioned cryptographic keys that can be used with the Web Cryptography API [31]. At the time of writing this thesis the WebCrypto Key Discovery specification was in the working draft phase which means that the document has been published for review by the community, including W3C members, the public, and other technical organizations. Ongoing and most up-to-date work

[17] The Browser Object Model (BOM) is browser-specific convention to allow the JavaScript to interact with the web browser

[18] Also known as Web Cryptography Key Discovery

can be found in the editor's draft. The document is intended to eventually become a W3C recommendation. The specification does not yet have any implementation available [78].

Web Cryptography API Use Cases is a Working Group Note[19] published by the Web Cryptography Working Group of the World Wide Web Consortium (W3C), which is a technical report that gives overview of the target use cases for a cryptographic API for the web. The use cases are described as scenarios and they represent some of the expected functionality that may be achieved by using the Web Cryptography API [31]. It presents the primary use cases, showing what the working group hopes to achieve first [35].

2.10 Security Considerations and Caveats

The Web Cryptography API does not change the fundamental Web Security model, which is based on the same-origin concept which restricts how a document or script loaded from one origin can interact with a resource from another origin [79;80]. In web cryptography, same-origin policy is used to limit the usage of (cryptography) keys to the same origin where the keys were originally generated.

Developers must be familar with existing threats to a web application and the underlying security model employed and take all possible steps for protecting the JavaScript run-time environment from threats. The Web Security model includes various access-control mechanisms, of which some are upcoming features that should be enabled for more secure JavaScript run-time environment:

- **HTTP Strict Transport Security standard (HSTS)** [80] which lets websites announce to browsers that they can be accessed only via HTTP Secure connection.
- **Content Security Policy (CSP)** [81] which informs the browser about the sources from which the application expects to load resources.
- **HTTP Public Key Pinning (HPKP)** [82] which allows web host operators to instruct user agents to enforce the usage of specific cryptographic identities over a period of time, effectively mitigating some form of Men-In-The-Middle attacks [83].
- **Subresource integrity (SRI)** [84] which allows the user agents to verify that a fetched resource has been delivered without unexpected manipulation.

[19] The Working Group may publish material that is not a formal specification as a working group note [30]

The Web Cryptography API is an low-level library that does not provide any default usage values. This is why usage of the Web Cryptography API requires that the developers are proficient in cryptographic literature and in using cryptographic primitives, know how to use them and know what weaknesses algorithms have if used inappropriately.

The mere use of cryptography does not automatically make a system secure and while cryptography is certainly useful, the security of the whole system must be considered. The most basic principle of security is that overall security of the system is no stronger than its weakest part [85].

Developers should note the following caveats when using the Web Cryptography API:

- CryptoKey with “encrypt” usage cannot be used to decrypt and a key with “decrypt” usage cannot be used to encrypt, as commonly used to perform digital signing of data (see section 2.2.3 on page 8). Instead a separate key pair with signing and verifying keys must be used to perform digital signing when using the API.
- Asymmetric cryptographic algorithms are not meant to be used to encrypt long messages. Usually the maximum message size is shorter than the key size used (for example a RSA-OAEP algorithm with 2048bit key-size allows a maximum of 1704 bits (or 213 bytes) of a message to be encrypted because of padding that is added to the message for security reasons).
- When the cryptographic keys are stored locally in the browser, an attacker could manage to steal the keys by injecting a malicious script by exploiting XSS (Cross-Site Scripting) [86] or other vulnerability in the application.

3 Case Study

3.1 Introduction

Keeping secrets truly secret is becoming harder each day [18]. When users want to store or share information online, they have currently very limited options for protecting their identity and the privacy of their confidential data. Each message sent using an instant messaging service or file stored in an online service could end up in wrong hands, as the service providers could be forced to give up the information by a court order or the data could also be stolen by foreign spying agencies or hackers [87].

One way to mitigate these kinds of privacy intrusions is to protect the data before it leaves the user's computer by encrypting the data with a strong algorithm and making sure that the plain text message or the secret keys are not transmitted to the service provider. If these precautions are taken, even if the service provider wanted to, the service owners could not decrypt and read the contents of the stored data [87].

For this thesis, as an case study, a proof-of-concept end-to-end secure example application for sharing encrypted and digitally signed messages was implemented. The example application is called **SecretNotes** and it utilizes the new Web Cryptography API for performing cryptographic operations directly on the web browser. The example application uses asymmetric public-key and symmetric secret-key cryptography as a means to protect the sent messages and also provides a digital signature that can be verified by the receiver (if the sender chooses to share identity). Symmetric secret-key cryptography is used to encrypt and decrypt the actual message payload, allowing it to be longer than the maximum length that could be normally encrypted using asymmetric cryptography.

The application does not reveal the plain text message, the decryption keys or other metadata such as the sender identity or the receiver's name to the server, so nothing can be leaked or revealed since server does not contain anything that could be used to decrypt the messages (traditionally instant messaging applications store the messages and decryption keys on the server which means that if the server security is compromised all of the messages and the decryption keys needed to decrypt the messages could be leaked). The only metadata that the example application shares with the server is the fingerprint, an SHA-1 message digest (hash) of the public-key which was used to encrypt the note contents. The fingerprint is used to index the notes on the server so that the notes can be retrieved by the user which have the correct decryption key for reading the notes. The note creation date is managed by the server and the note is automatically expired after 24 hours have passed.

The example application utilizes two helper modules, called utility module (utils.js) and cryptography module (cryptography.js) which were written for this project. The utility module provide methods for handling input and output data in a various formats that the *SubtleCrypto* interface methods require. The cryptography module provide high-level methods for performing cryptographic operations. For example, in order to encrypt and at the same time digitally sign plain text message the cryptography module provides *encryptAndSign* method that can be utilized. The complete source code for the example application, example server implementation and the helper modules can be found in the GitHub repository [70].

The example implementation use concept of **identities** and enable the user to create new private identities and to import private or public identities. Identity can have two sides, the private and the public side. When identity is imported from shared public identity the identity only contains the public side. Private identity contains the private cryptographic keys needed to decrypt and digitally sign sent notes. Public identity contains the public cryptographic keys needed to encrypt notes and verify digital signatures. The Public identity can be shared freely but private identity should be kept secret.

The *SecretNotes* application implements and enables the following features:

- **Managing of User Identity**:
  - **Creation** of identity, by allowing the user to generate a public and private key-pair for encrypting and decrypting data, and signing and verifying a key-pair for creation of a digital signature and verification.

  - **Sharing** of user identity, by allowing the user to export the public-side of their own identity and share it safely with other user using any unsecure or public channel. Shared user identity enables other users to send messages to the user which the shared identity belongs to.
  - **Backup** of user identity, by allowing the user to export the private and public-side of the identity for safe-keeping or moving to an other computer or device.
- **Sending and Receiving Notes**:
  - **Encryption** of a message, by allowing the user to encrypt the sent message using the recipient public-key, so that only the recipient can read the message.
  - **Decryption** of a message, by allowing the user to decrypt the received message using the personal private-key.
- **Digital Signing**:
  - **Signing** the sent message with a digital signature, by allowing the user to create digital signature using a personal signing-key.
  - **Verifying** the identity of the sender, by allowing the user to verify the digital signature and store, and manage verifying keys of the sender.
  - **Anonymous** identity by choosing not to include sender's user identity information in the message.

3.2 Sending and Receiving Digitally Signed Messages

In this section, technical process of sending and receiving digitally signed messages is presented.

Prerequisites:

- For the sender to be able to send message to the receiver, the sender needs to have the receiver's *public-key* for encrypting the message.
- For the receiver to be able to verify the sender's digital signature embedded in the message, the receiver needs to have the sender's *verify-key* for verifying the digital signature.

When the sender (lets call the sender *Alice*) sends an encrypted and digitally signed message (called *Note*) to the receiver (lets call the receiver *Bob*), the following steps are required.

Figure 8 below shows the steps used to encrypt and sign the message before it is sent to the receiver. The steps are specified below.

1. Alice writes a plaintext message.
2. The *plaintext* is signed with Alice's *signing-key*.
3. Alice generates a random *symmetric-key* and
4. a random *initialization vector (IV)*.
5. Encrypts them both with Bob's *public-key*.
6. Encrypts the *plaintext* and the digital signature form (step 2) with the *symmetric-key* and *initialization vector* (steps 3. and 4.) and
7. combines resulting *ciphertexts* into one data package that can be sent to the receiver.

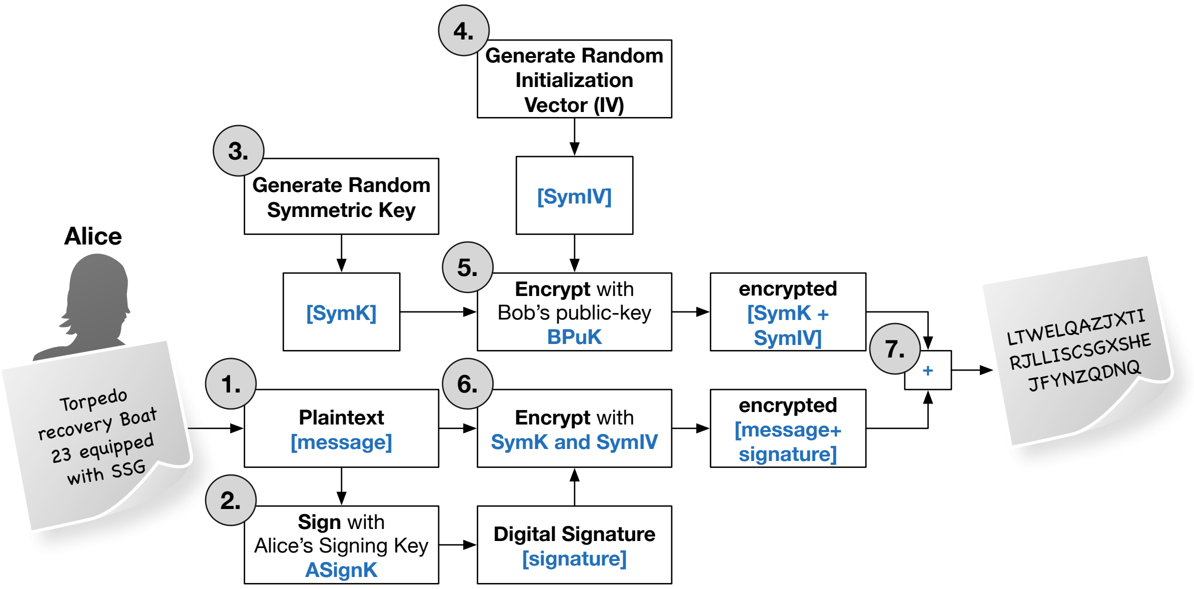

4.
Generate Random Initialization Vector (IV)
[SymIV]
3.
Generate Random Symmetric Key
[SymK]
5.
Encrypt with Bob's public-key BPuK
encrypted [SymK + SymIV]
7.
+
LTWELQAZJXTI RJLLISCSGXSHE JFYNZQDNQ
Alice
Torpedo recovery Boat 23 equipped with SSG
1.
Plaintext [message]
6.
Encrypt with SymK and SymIV
encrypted [message+ signature]
2.
Sign with Alice's Signing Key ASignK
Digital Signature [signature]

Figure 8: Encrypting and signing a message (Copyright © 2015 Mika Luoma-aho)

Figure 9 on the following page shows the steps used to decrypt and verify the digital signature on the message when it is received from the sender. The steps are specified below.

1. Bob splits the input data to its known components: encrypted symmetric-key and the initization vector (IV) *[SymK and SymIV]*, encrypted message *[message]* and the digital signature *[signature]*.
2. Bob decrypts the *[SymK+SymIV]* data with Bob's *private-key*, revealing *symmetric-key* and the *initization vector (IV)* to be used with the key.
3. Bob decrypts the *[plaintext+digitalsignature]* data with the *symmetric-key* and the *initialization vector (IV)*, revealing the *plaintext* and *digital signature*.

4. Bob verifies *digitalsignature* using the *plaintext* and Alice's *verify-key*.
5. Bob can read the *plaintext* and can be sure that the sender was Alice because the *digital signature* was verified (step 6).

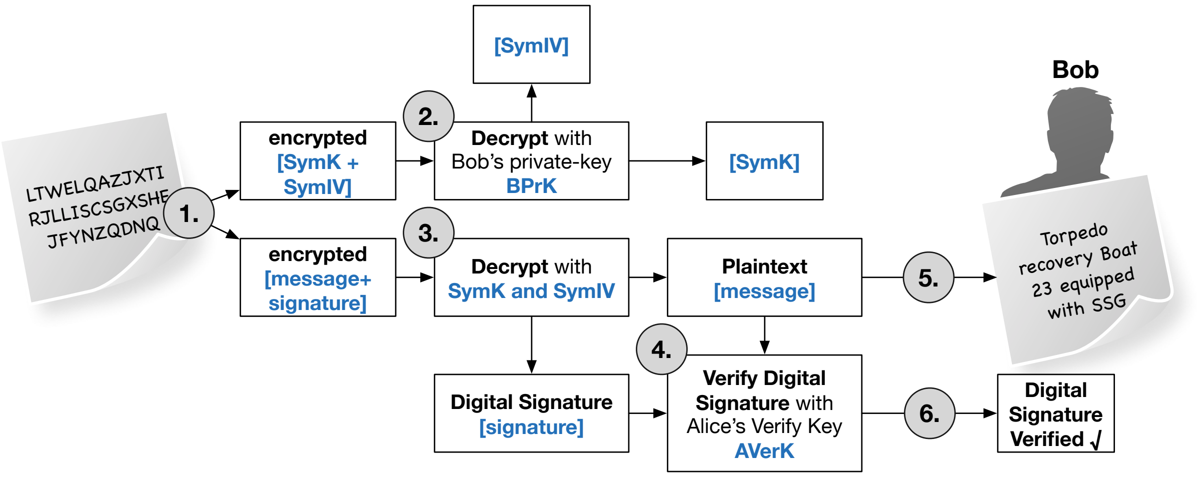

LTWELQAZJXTI
RJLLISCSGXSHE
JFYNZQDNQ
1.
encrypted [SymK + SymIV]
2.
Decrypt with Bob's private-key BPrK
[SymIV]
[SymK]
Bob
encrypted [message+ signature]
3.
Decrypt with SymK and SymIV
Plaintext [message]
5.
Torpedo recovery Boat 23 equipped with SSG
Digital Signature [signature]
4.
Verify Digital Signature with Alice's Verify Key AVerK
6.
Digital Signature Verified √

Figure 9: Decrypting and verifying a message (Copyright © 2015 Mika Luoma-aho)

In the case study example application, when the sender decides to share identity and digitally sign the message, the sender *public-key* and *verify-key* is attached with the message and encrypted with the receiver *public-key*. Figure 10 shows the final data package contents used by the example application when the sender has digitally signed the message. In the figure, the *Message* is the plaintext message to be sent, *DigitalSignature* contains the output of the sign operation, APuK is the sender's public-key, AVerK is the sender's verify-key, and the *SymK* is the symmetric key that is used to encrypt the previously mentioned information. Finally the SymK (symmetric-key) and the SymIV (initialization vector for symmetric-key) are encrypted with the receiver's public-key (BPuK).

$$[Message + DigitalSignature + APuK + AVerK]_{SymK} + [SymK + SymIV]_{BPuK}$$

Figure 10: Encrypted and digitally signed message.

The example implementation has some shortcomings like the fact that the same public-key is used over and over again to encrypt messages. This means that an attacker could record all of the messages over some extended period of time, and then if the private-key is compromised at any point in the future, the attacker could decrypt all of the previously sent messages too. Some new modern cryptographic protocols like OTR [36] remedy this situation by using a short lived key exchange protocol like the Diffie–Hellman key exchange [88] which allows two parties to securely share cryptographic keys over a public channel [89]. Since these key exchanges are short lived and the keys are temporal, recording the messages over time does not help the attacker, since there is no key that could be compromised in the future. Also since the keys are not stored anywhere, the messages can’t be opened even if the user device is stolen or otherwise compromised. This property is often called Perfect Forward Secrecy [37].

Possible improvements that could be made to the application are listed below:

- To increase privacy, the server could store all of the notes without any metadata at all, and the fingerprint used to identity the receiver could be replaced with some random identifier, which would mean that clients would have to search through all of the data in order to find notes that they can decrypt. Alternatively, the receiver and sender could share a secret passphrase and the message digest of that passphrase could be used as a index for the messages stored on the server.
- To increase privacy even more, the server and client could randomly store additional random notes that cannot be decrypted by anybody, but are used to confuse and slow possible attackers since the attackers do not know which messages contain real data and which do not.

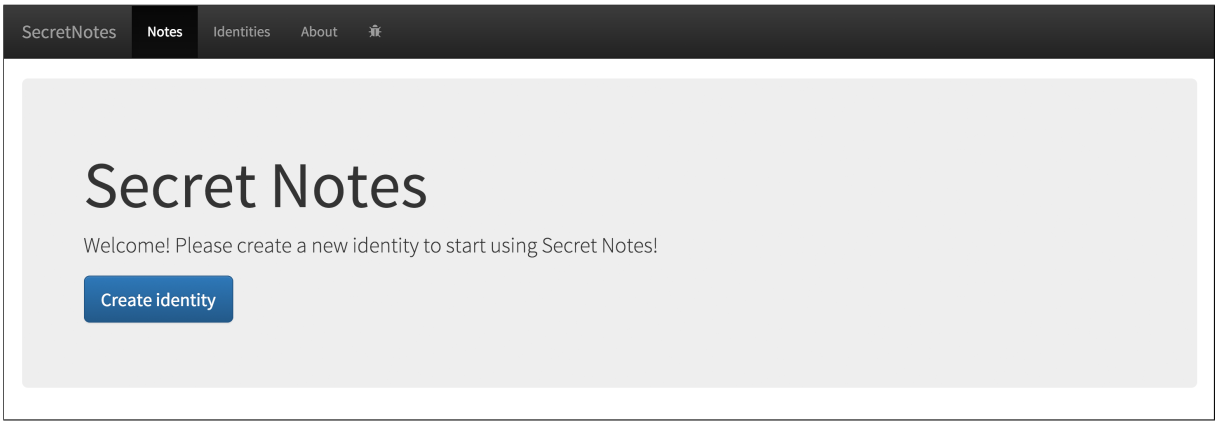
Possible issues with the implementation:

- It is possible that the database used for storing the CryptoKey objects could be deleted or cleared, so developers should allow the user to export the public and private identity for safe keeping.
- Keys have to be shared and managed manually.
- User identity do not have any master password, meaning that anybody using the same browser and web application can access and utilize the keys freely.

3.3 Usage

In this section, a simple use case using the example application is presented. The use case scenario is the following: **Alice** is going to send a digitally signed note to **Bob**. First, *Alice* and *Bob* is going to generate identities on their own computers. Then *Bob* exports public identity and shares it with *Alice* so that *Alice* can send note to *Bob* using *Bob's* public cryptographic key. Since Alice uses *Bob's* public cryptographic key to encrypt the sent note, only *Bob* can open and read the note. *Alice* is also going to digitally sign the sent note and include *Alice's* own public identity with the note so that *Bob* can reply to *Alice's* note and also verify that the digital signature is valid (providing both authentication and non-repudiation) and the note has not been tampered with (providing integrity).

First, when *Alice* opens the application in a web browser, the view as seen in figure 11 is shown. Since *Alice* has not created an identity, the application instructs to start by creating a new identity.



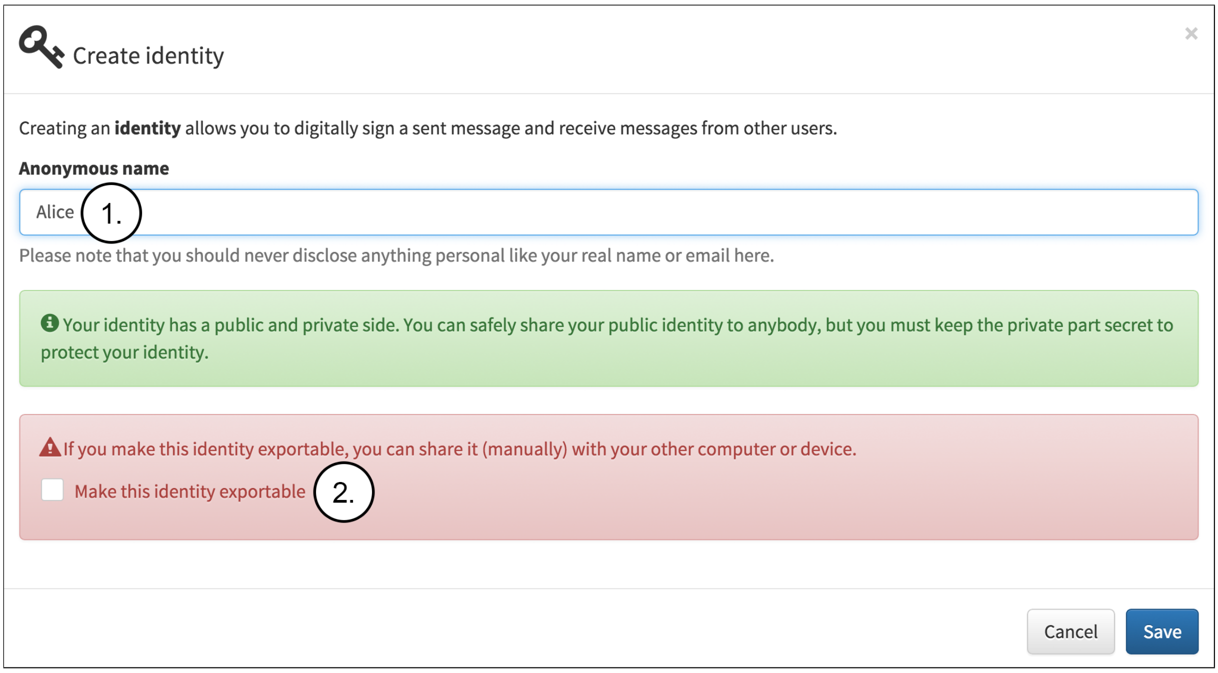
Secret Notes

Welcome! Please create a new identity to start using Secret Notes!

Figure 11: Welcome view of the application (Screenshot of the example application)

Alice chooses to create a new identity as seen in figure 12. The figure contains the following notes,

1. *Alice* fills in the name which will be used internally for this identity. The name is not shared and is only used for identifying this identity stored locally.
2. *Alice* decides not to make the identity exportable which means that *Alice's* private identity cannot be exported out of the application. This option gives a extra security since *Alice's* private key is not exportable from the JavaScript environment.



Create identity

Creating an **identity** allows you to digitally sign a sent message and receive messages from other users.

Anonymous name

Alice

Please note that you should never disclose anything personal like your real name or email here.

Your identity has a public and private side. You can safely share your public identity to anybody, but you must keep the private part secret to protect your identity.

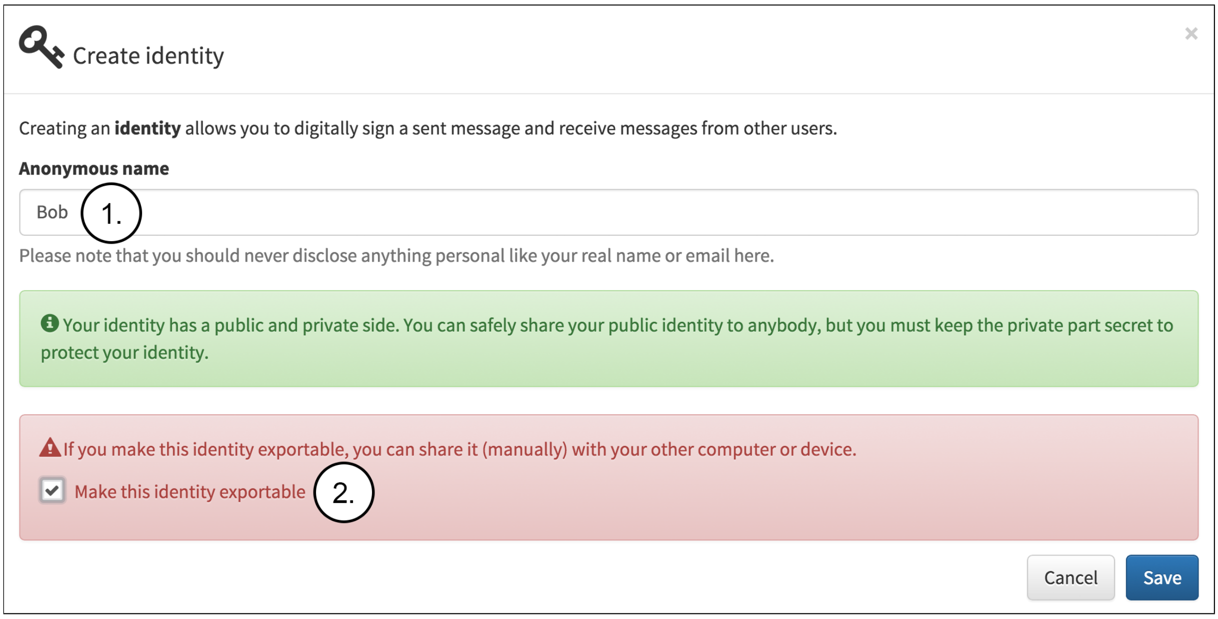
If you make this identity exportable, you can share it (manually) with your other computer or device.

□ Make this identity exportable

Figure 12: Creating a new identity (Screenshot of the example application)

Next, *Bob* creates an identity on his own computer as seen in figure 12 on the preceding page. The figure contains the following notes,

1. *Bob* fills in the name which will be used internally for this identity. The name is not shared and is only used for identifying this identity stored locally.
2. *Bob* decides to make the identity exportable which means that *Bob* can share his private identity with other devices or computers that he also uses for sending notes.



Create identity

Creating an **identity** allows you to digitally sign a sent message and receive messages from other users.

Anonymous name

Bob

Please note that you should never disclose anything personal like your real name or email here.

Your identity has a public and private side. You can safely share your public identity to anybody, but you must keep the private part secret to protect your identity.

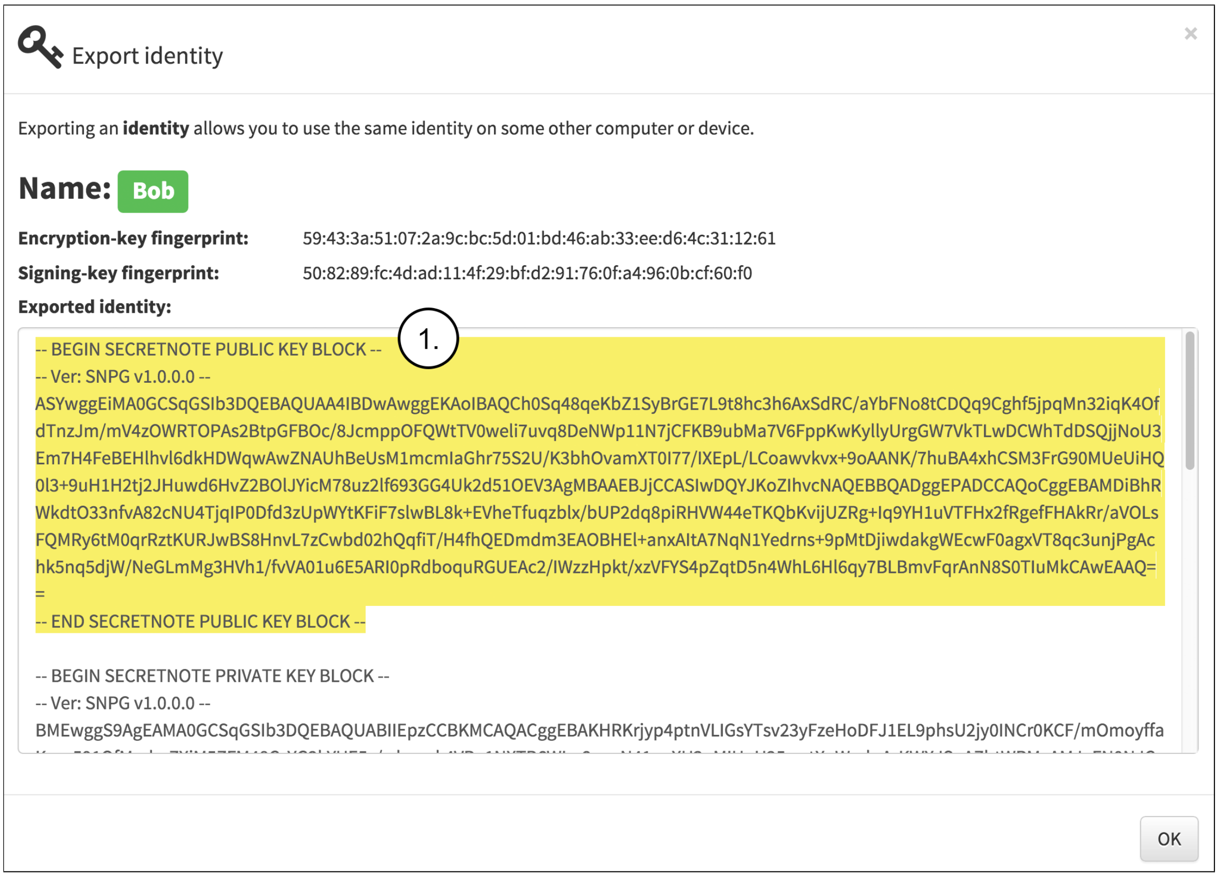
If you make this identity exportable, you can share it (manually) with your other computer or device.

☑ Make this identity exportable

Figure 13: Creating a new identity (Screenshot of the example application)

Bob exports the public identity as seen in figure 14. The figure contains the following note,

1. *Bob* selects and copies the public part of the identity for sharing it with *Alice*. *Bob* can share the public identity using any public or private communication channel.



Exporting an **identity** allows you to use the same identity on some other computer or device.

Name: Bob

Encryption-key fingerprint: 59:43:3a:51:07:2a:9c:bc:5d:01:bd:46:ab:33:ee:d6:4c:31:12:61

Signing-key fingerprint: 50:82:89:fc:4d:ad:11:4f:29:bf:d2:91:76:0f:a4:96:0b:cf:60:f0

Exported identity:

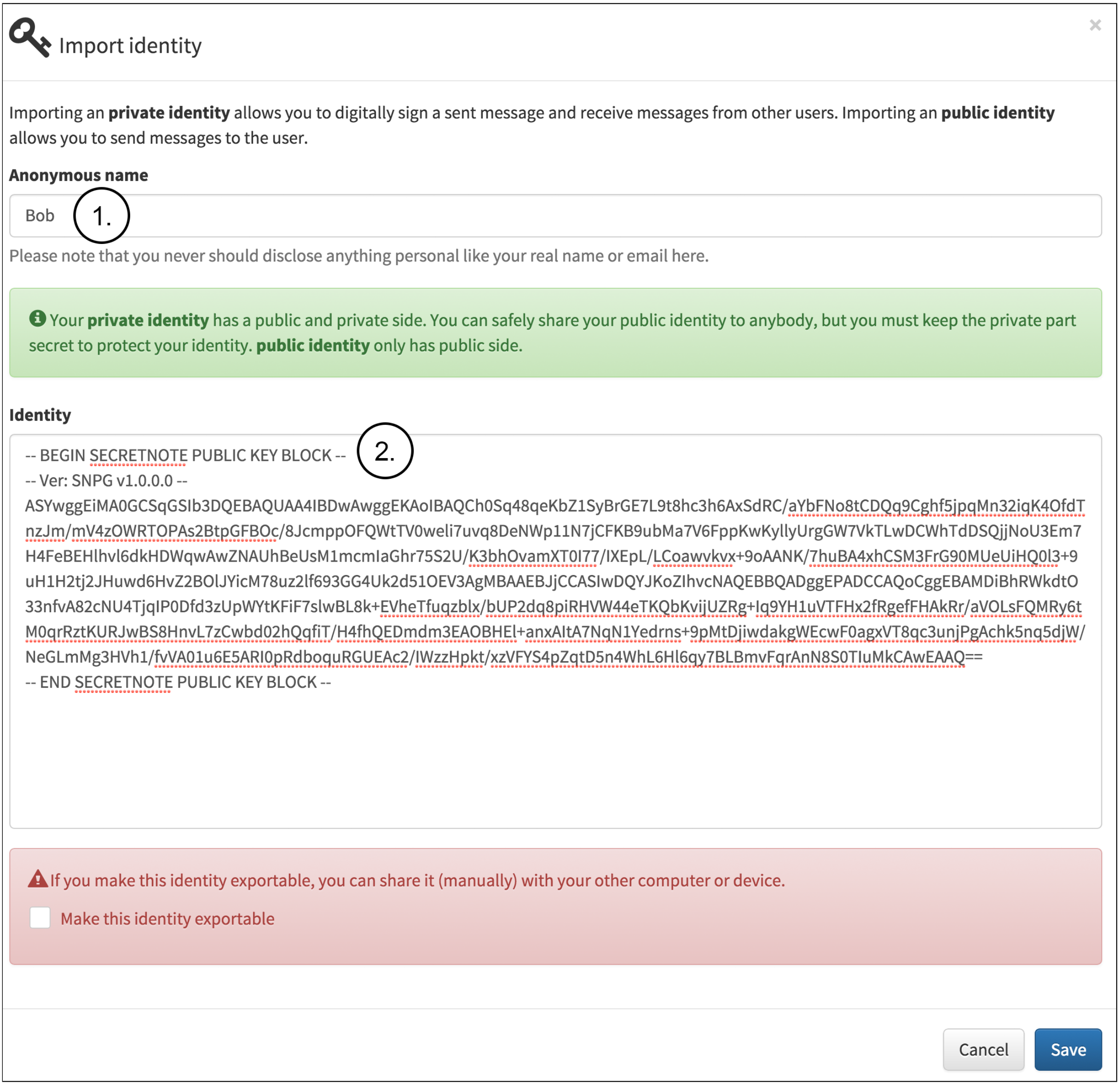
```
-- BEGIN SECRETNOTE PUBLIC KEY BLOCK --
-- Ver: SNPG v1.0.0.0 --
ASYwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCh0Sq48qeKbZ1SyBrGE7L9t8hc3h6AxSdRC/aYbFNo8tCDQq9Cghf5jpqMn32iqK4Of
dTnzJm/mV4zOWRTOPAs2BtpGFBOc/8JcmppOFQWtTV0weli7uvq8DeNWp11N7jCFKB9ubMa7V6FppKwKyllyUrgGW7VkTLwDCWhTdDSQjjNoU3
Em7H4FeBEHlhvl6dkHDWqwAwZNAUhBeUsM1mcmIaGhr75S2U/K3bhOvamXT0I77/IXEpL/LCoawvkvx+9oAANK/7huBA4xhCSM3FrG90MUeUiHQ
0l3+9uH1H2tj2JHuwd6HvZ2BOlJYicM78uz2lf693GG4Uk2d51OEV3AgMBAAEBJjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMDiBhR
WkdtO33nfvA82cNU4TjqIP0Dfd3zUpWYtKFiF7slwBL8k+EVheTfuqzblx/bUP2dq8piRHVW44eTKQbKvijUZRg+Iq9YH1uVTFHx2fRgefFHAkRr/aVOLs
FQMRy6tM0qrRztKURJwBS8HnvL7zCwbd02hQqfiT/H4fhQEDmdm3EAOBHEl+anxAItA7NqN1Yedrns+9pMtDjiwdakgWEcwF0agxVT8qc3unjPgAc
hk5nq5djW/NeGLmMg3HVh1/fvVA01u6E5ARI0pRdboquRGUEAc2/IWzzHpkt/xzVFYS4pZqtD5n4WhL6Hl6qy7BLBmvFqrAnN8S0TIuMkCAwEAAQ=
=
-- END SECRETNOTE PUBLIC KEY BLOCK --

-- BEGIN SECRETNOTE PRIVATE KEY BLOCK --
-- Ver: SNPG v1.0.0.0 --
BMEwggS9AgEAMA0GCSqGSIb3DQEBAQUABIIEpzCCBKMCAQACggEBAKHRKrjyp4ptnVLIGsYTsv23yFzeHoDFJ1EL9phsU2jy0INCr0KCF/mOmoyffa
```

Figure 14: Exporting an identity (Screenshot of the example application)

Next, *Alice* has received *Bob's* public identity by email, web page or any other public communication channel and imports it as a known identity as seen in figure 15. The figure contains the following notes,

1. *Alice* fills in the name which will be used internally for this identity.
2. *Alice* copies and pastes *Bob's* public identity into the identity field.



Import identity

Importing an **private identity** allows you to digitally sign a sent message and receive messages from other users. Importing an **public identity** allows you to send messages to the user.

Anonymous name

Bob

Please note that you never should disclose anything personal like your real name or email here.

Your **private identity** has a public and private side. You can safely share your public identity to anybody, but you must keep the private part secret to protect your identity. **public identity** only has public side.

Identity

```
-- BEGIN SECRETNOTE PUBLIC KEY BLOCK --
-- Ver: SNPG v1.0.0.0 --
ASYwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCh0Sq48qeKbZ1SyBrGE7L9t8hc3h6AxSdRC/aYbFNo8tCDQq9Cghf5jpqMn32iqK4OfdT
nzJm/mV4zOWRTOPAs2BtpGFBOc/8JcmppOFQWtTV0weli7uvq8DeNWp11N7jCFKB9ubMa7V6FppKwKyllyUrgGW7VkTLwDCWhTdDSQjjNoU3Em7
H4FeBEHlhvl6dkHDWqwAwZNAUhBeUsM1mcmIaGhr75S2U/K3bhOvamXT0I77/IXEpL/LCoawvkvx+9oAANK/7huBA4xhCSM3FrG90MUeUiHQ0l3+9
uH1H2tj2JHuwd6HvZ2BOlJYicM78uz2lf693GG4Uk2d51OEV3AgMBAAEBJjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMDiBhRWkdtO
33nfvA82cNU4TjqIP0Dfd3zUpWYtKFiF7slwBL8k+EVheTfuqzblx/bUP2dq8piRHVW44eTKQbKvijUZRg+Iq9YH1uVTFHx2fRgefFHAkRr/aVOLsFQMRy6t
M0qrRztKURJwBS8HnvL7zCwbd02hQqfiT/H4fhQEDmdm3EAOBHEl+anxAItA7NqN1Yedrns+9pMtDjiwdakgWEcwF0agxVT8qc3unjPgAchk5nq5djW/
NeGLmMg3HVh1/fvVA01u6E5ARI0pRdboquRGUEAc2/IWzzHpkt/xzVFYS4pZqtD5n4WhL6Hl6qy7BLBmvFqrAnN8S0TIuMkCAwEAAQ==
-- END SECRETNOTE PUBLIC KEY BLOCK --
```

If you make this identity exportable, you can share it (manually) with your other computer or device.

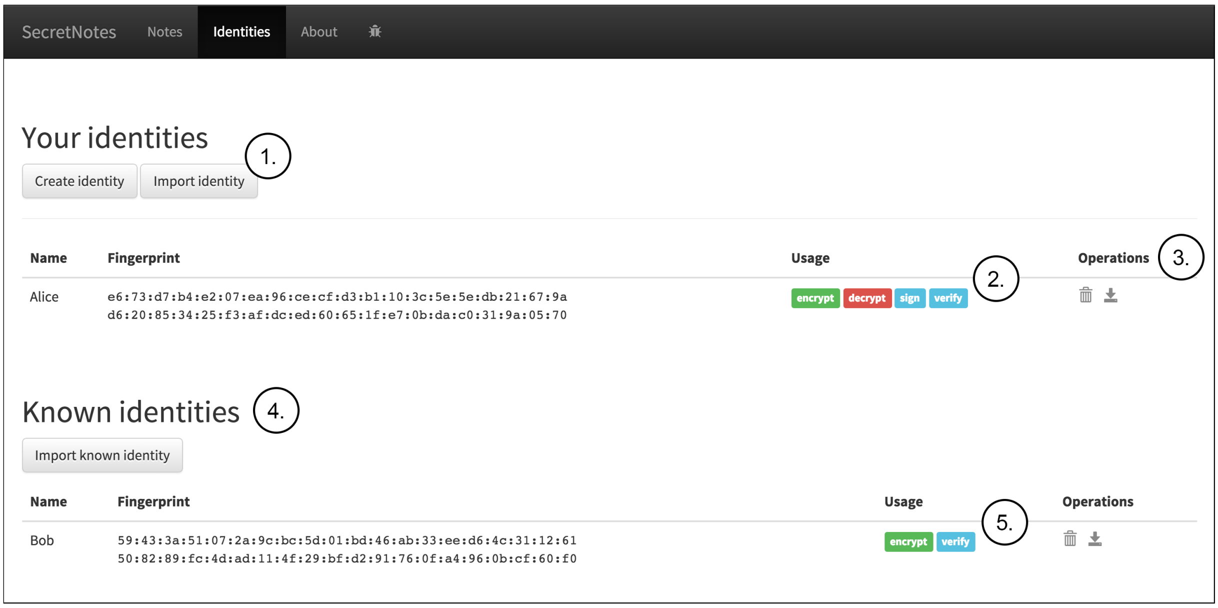
☐ Make this identity exportable

Cancel Save

Figure 15: Importing a identity (Screenshot of the example application)

Figure 16 shows how *Alice* can manage identities stored in the browser by using the identities section of the application. The figure contains the following notes,

1. Your identities section contains the private identities and allows the user to create new identity and import identities.
2. Usage section displays the usage information for identity. In this example *Alice's* own private identity can be used for encrypting, decrypting, signing and verifying.
3. Identities can be deleted with the trashcan icon and exported with the download icon.
4. Known identities section contains the public identities that have been imported. In this case *Bob's* public identity is listed.
5. Usage section displays the usage information for identity. In this example *Bob* identity can be used for encrypting and verifying.



Your identities

Name	Fingerprint	Usage	Operations
Alice	e6:73:d7:b4:e2:07:ea:96:ce:cf:d3:b1:10:3c:5e:5e:db:21:67:9a d6:20:85:34:25:f3:af:dc:ed:60:65:1f:e7:0b:da:c0:31:9a:05:70	encrypt decrypt sign verify	🗑 ⬇

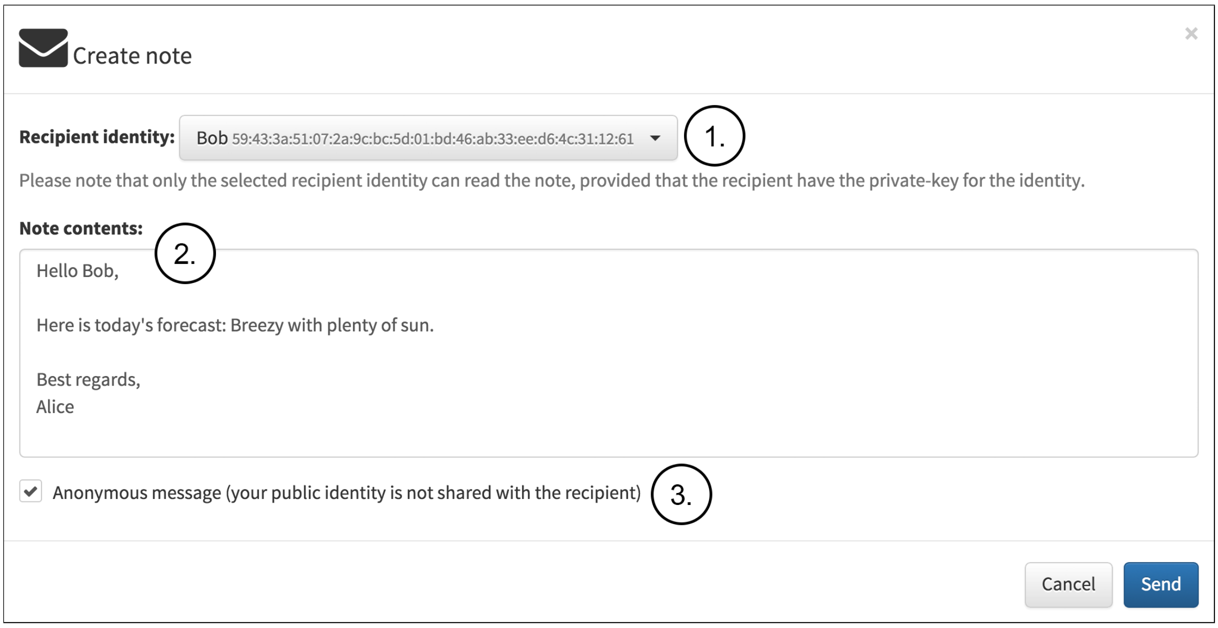
Known identities

Name	Fingerprint	Usage	Operations
Bob	59:43:3a:51:07:2a:9c:bc:5d:01:bd:46:ab:33:ee:d6:4c:31:12:61 50:82:89:fc:4d:ad:11:4f:29:bf:d2:91:76:0f:a4:96:0b:cf:60:f0	encrypt verify	🗑 ⬇

Figure 16: Managing identities (Screenshot of the example application)

Next, *Alice* decides to write a note to Bob as seen in figure 17. The figure contains the following notes,

1. *Alice* selects Bob as the target identity from the drop down list.
2. *Alice* writes the note to be sent.
3. *Alice* decides to remain anonymous. Thus she will not reveal her true identity to the receiver.



Create note

Recipient identity: Bob 59:43:3a:51:07:2a:9c:bc:5d:01:bd:46:ab:33:ee:d6:4c:31:12:61

1.

Please note that only the selected recipient identity can read the note, provided that the recipient have the private-key for the identity.

Note contents:

2.

Hello Bob,

Here is today's forecast: Breezy with plenty of sun.

Best regards,
Alice

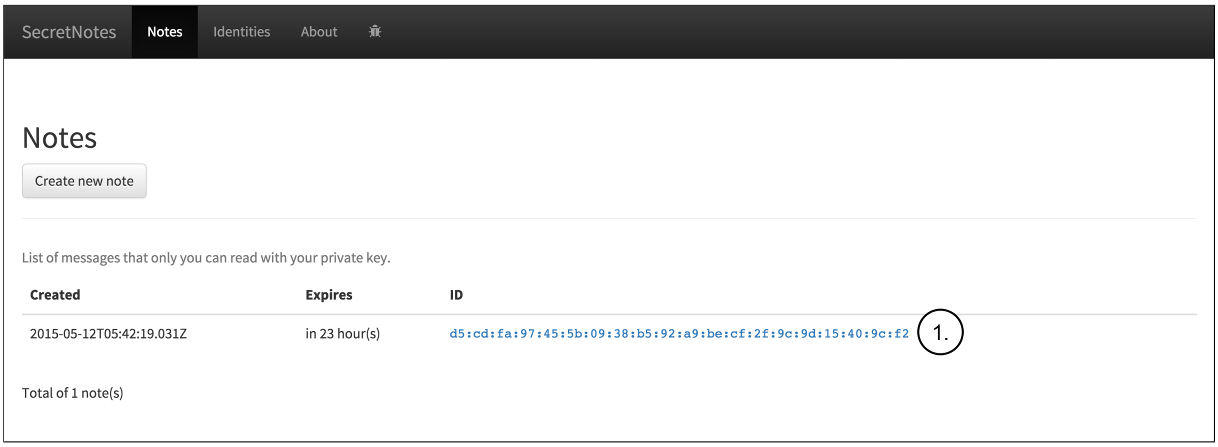
☑ Anonymous message (your public identity is not shared with the recipient)

3.

Figure 17: Creating a new note (Screenshot of the example application)

Bob receives the note as can be seen in figure 18. The figure contains the following note,

1. The only info about the received note that *Bob* can see is the note creation time, when the note is going to expire and the ID which is message digest (hash) which is calculated from the encrypted note.



Notes

List of messages that only you can read with your private key.

Created	Expires	ID
2015-05-12T05:42:19.031Z	in 23 hour(s)	d5:cd:fa:97:45:5b:09:38:b5:92:a9:be:cf:2f:9c:9d:15:40:9c:f2

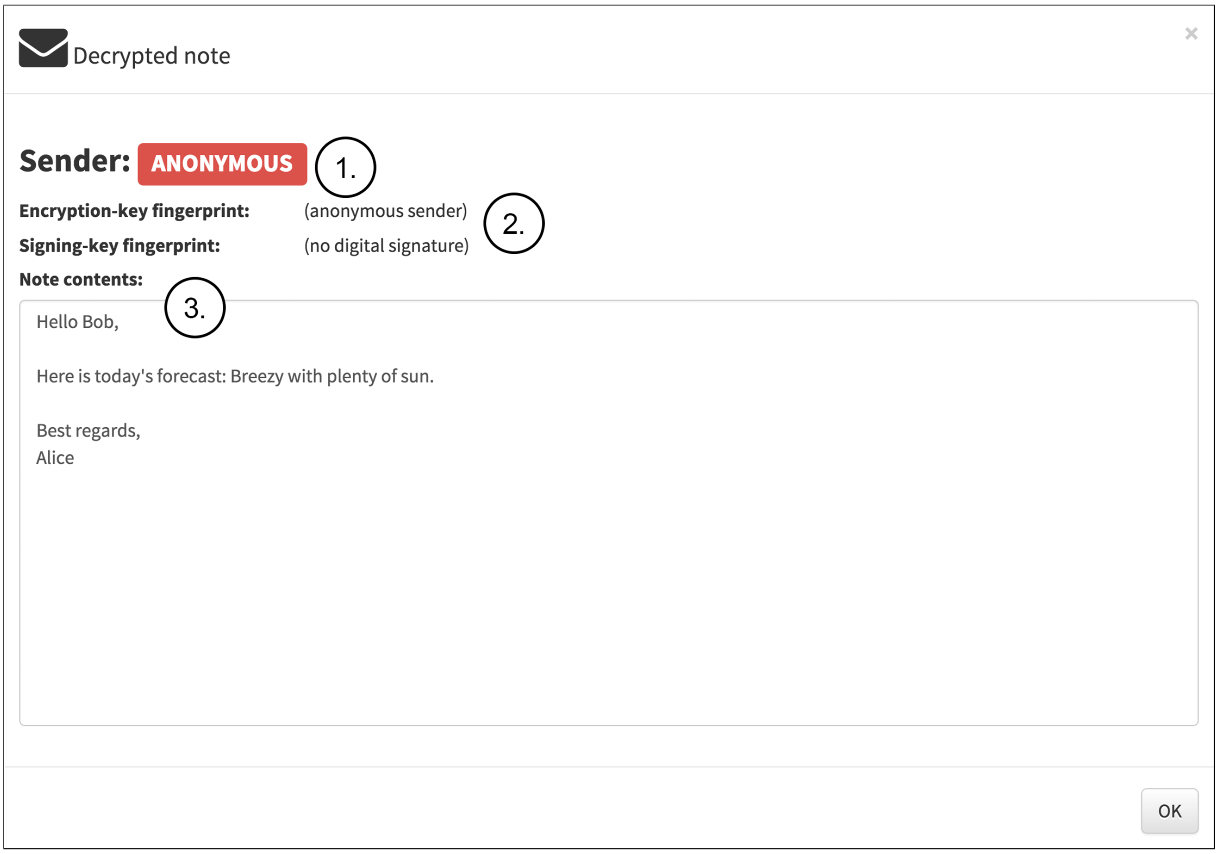
Total of 1 note(s)

Figure 18: Notes view (Screenshot of the example application)

Next, *Bob* decrypts the note sent by *Alice* as seen in figure 19. The figure contains the following notes,

1. Since the received note was sent as anonymous there is no way of really knowing who sent the message.
2. *Encryption-key* and *Signing-key fingerprint* values are empty since the note was anonymously sent.
3. The note content is visible since it was successfully decrypted using *Bob's* private-key.

Bob can now call or meet *Alice* and ask if *Alice* really sent the note.



Sender: ANONYMOUS

Encryption-key fingerprint: (anonymous sender)

Signing-key fingerprint: (no digital signature)

Note contents:

Hello Bob,

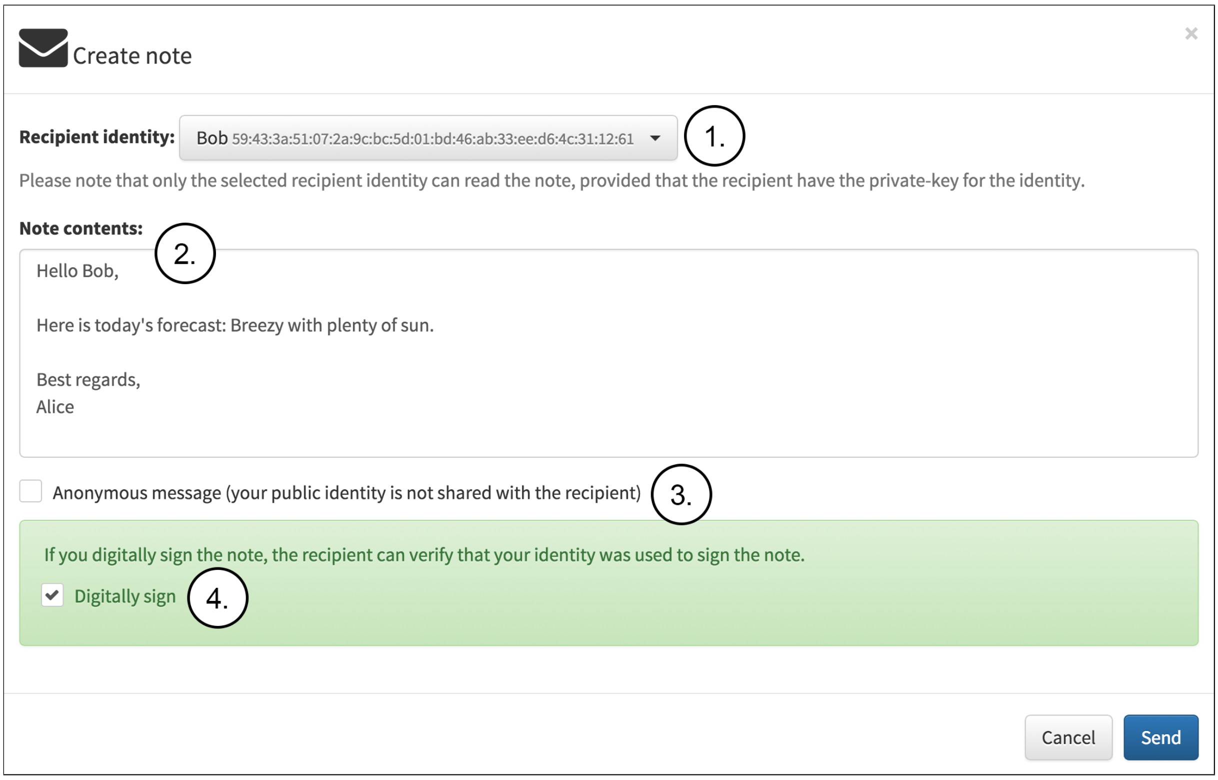
Here is today's forecast: Breezy with plenty of sun.

Best regards,
Alice

Figure 19: Decrypted note (Screenshot of the example application)

Next, *Alice* decides to write another note to Bob as seen in figure 20. This time *Alice* is going to include public identity and also sign the note with a digital signature so that *Bob* can verify that note really was sent by *Alice*. The figure contains the following notes,

1. *Alice* selects *Bob* as the target identity from the drop down list.
2. *Alice* writes the note to be sent.
3. *Alice* decides to share identity to the receiver.
4. *Alice* decides to sign the note with digital signature.



Recipient identity: Bob 59:43:3a:51:07:2a:9c:bc:5d:01:bd:46:ab:33:ee:d6:4c:31:12:61

Please note that only the selected recipient identity can read the note, provided that the recipient have the private-key for the identity.

Note contents:

Hello Bob,

Here is today's forecast: Breezy with plenty of sun.

Best regards,
Alice

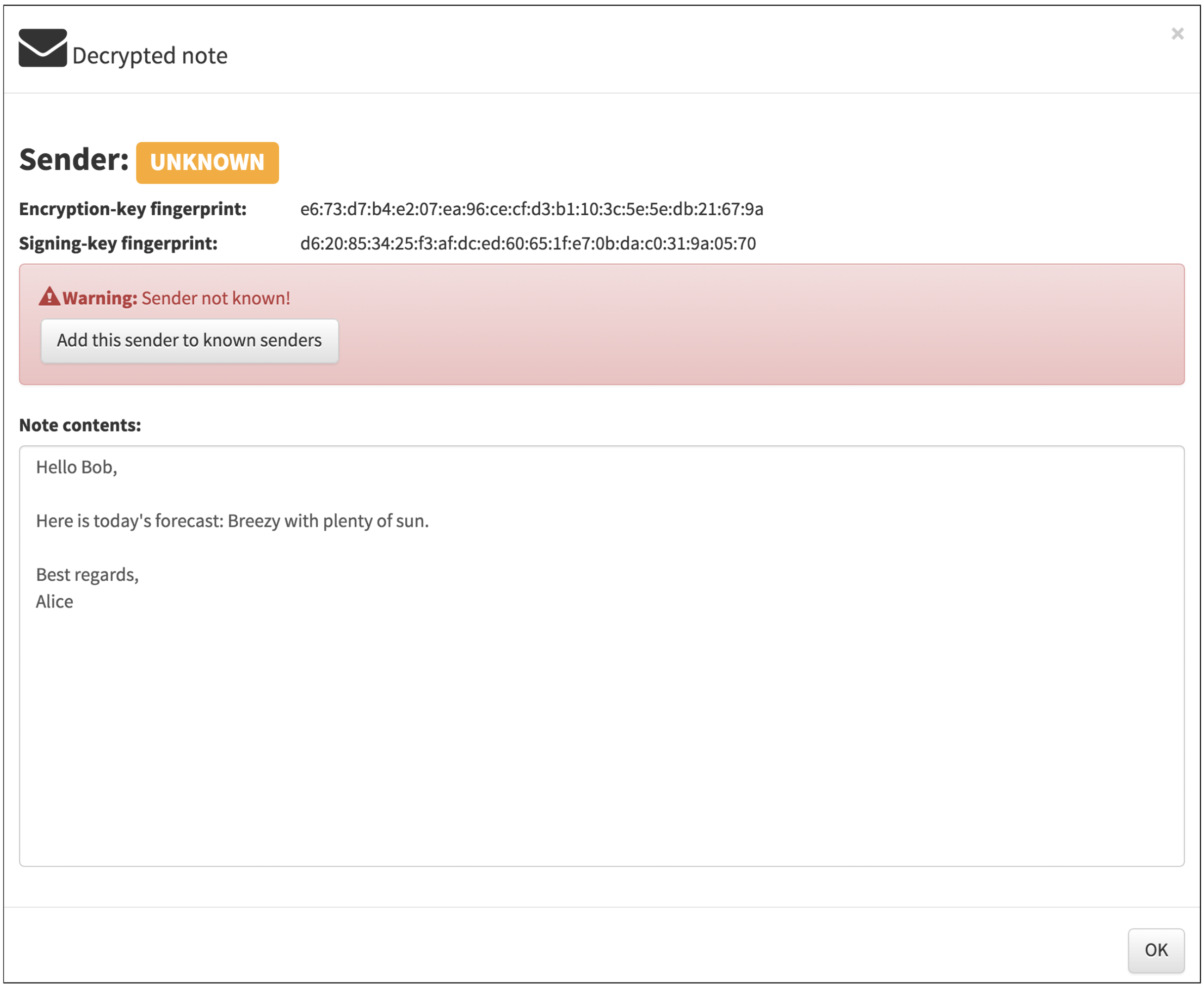
□ Anonymous message (your public identity is not shared with the recipient)

If you digitally sign the note, the recipient can verify that your identity was used to sign the note.

☑ Digitally sign

Figure 20: Creating a new note (Screenshot of the example application)

Bob receives another note, this time with *Alice's* public identity and also *Alice's* digitally signature as seen in figure 21 but the sender is still shown as *unknown* because *Bob* do not have *Alice's* identity. *Bob* should now verify that the encryption-key fingerprint (e6:73:d7:b4:e2:07:ea:96:ce:cf:d3:b1:10:3c:5e:5e:db:21:67:9a) belongs to *Alice*. After *Bob* have verified the fingerprint *Bob* can import the identity which was included with the note.



Sender: UNKNOWN

Encryption-key fingerprint: e6:73:d7:b4:e2:07:ea:96:ce:cf:d3:b1:10:3c:5e:5e:db:21:67:9a

Signing-key fingerprint: d6:20:85:34:25:f3:af:dc:ed:60:65:1f:e7:0b:da:c0:31:9a:05:70

Warning: Sender not known!

Note contents:

Hello Bob,

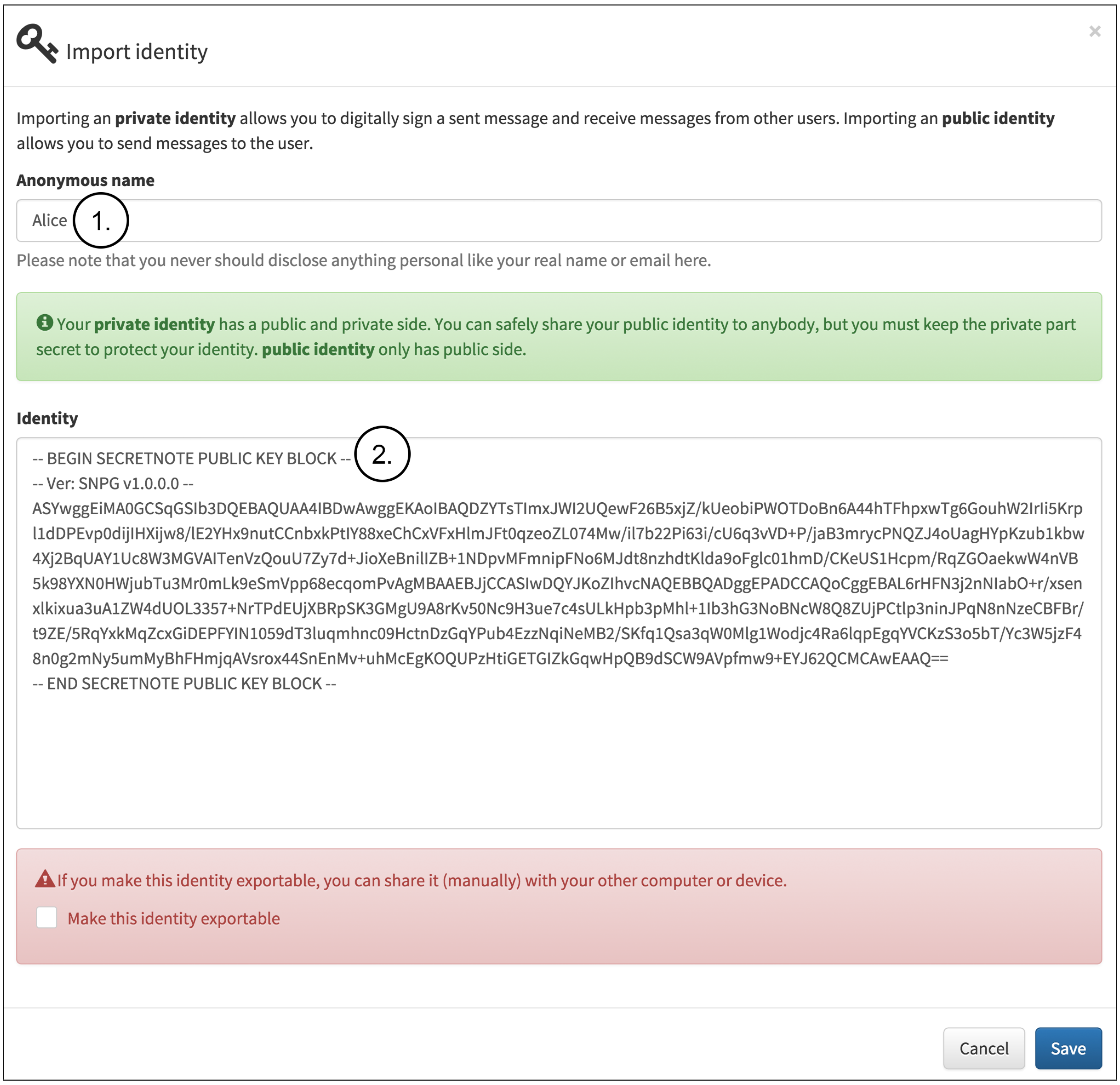
Here is today's forecast: Breezy with plenty of sun.

Best regards,
Alice

Figure 21: Received note from unknown sender (Screenshot of the example application)

Bob imports *Alice's* identity that he received with the previous note as seen in figure 22. The figure contains the following notes,

1. *Bob* fills in the name which will be used internally for this known public identity. The name is not shared and is only used for identifying this identity stored locally.
2. The public identity is automatically filled in from the received message.



Importing an **private identity** allows you to digitally sign a sent message and receive messages from other users. Importing an **public identity** allows you to send messages to the user.

Anonymous name

Alice

Please note that you never should disclose anything personal like your real name or email here.

Your **private identity** has a public and private side. You can safely share your public identity to anybody, but you must keep the private part secret to protect your identity. **public identity** only has public side.

Identity

```
-- BEGIN SECRETNOTE PUBLIC KEY BLOCK --
-- Ver: SNPG v1.0.0.0 --
ASYwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDZYTsTImxJWI2UQewF26B5xjZ/kUeobiPWOTDoBn6A44hTFhpxwTg6GouhW2IrIi5Krp
l1dDPEvp0dijIHXijw8/lE2YHx9nutCCnbxkPtIY88xeChCxVFxHlmJFt0qzeoZL074Mw/il7b22Pi63i/cU6q3vVD+P/jaB3mrycPNQZJ4oUagHYpKzub1kbw
4Xj2BqUAY1Uc8W3MGVAITenVzQouU7Zy7d+JioXeBnilIZB+1NDpvMFmnipFNo6MJdt8nzhdtKlda9oFglc01hmD/CKeUS1Hcpm/RqZGOaekwW4nVB
5k98YXN0HWjubTu3Mr0mLk9eSmVpp68ecqomPvAgMBAAEBJjCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL6rHFN3j2nNIabO+r/xsen
xlkixua3uA1ZW4dUOL3357+NrTPdEUjXBRpSK3GMgU9A8rKv50Nc9H3ue7c4sULkHpb3pMhl+1Ib3hG3NoBNcW8Q8ZUjPCtlp3ninJPqN8nNzeCBFBr/
t9ZE/5RqYxkMqZcxGiDEPFYIN1059dT3luqmhnc09HctnDzGqYPub4EzzNqiNeMB2/SKfq1Qsa3qW0Mlg1Wodjc4Ra6lqpEgqYVCKzS3o5bT/Yc3W5jzF4
8n0g2mNy5umMyBhFHmjqAVsrox44SnEnMv+uhMcEgKOQUPzHtiGETGIZkGqwHpQB9dSCW9AVpfmw9+EYJ62QCMCAwEAAQ==
-- END SECRETNOTE PUBLIC KEY BLOCK --
```

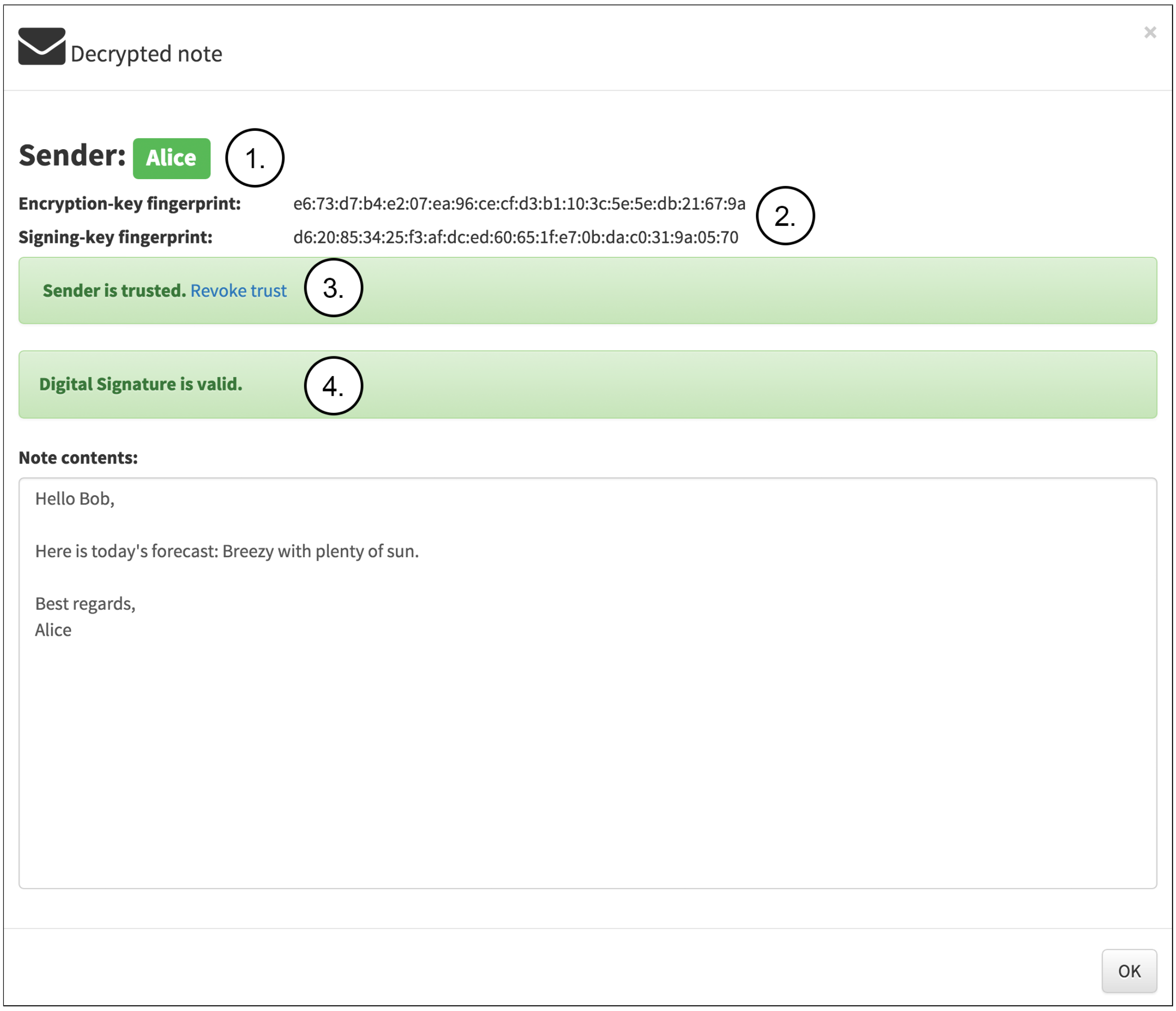
If you make this identity exportable, you can share it (manually) with your other computer or device.

□ Make this identity exportable

Figure 22: Importing an identity (Screenshot of the example application)

Now *Bob* can be sure that the sender really is *Alice* and that the message has not been tampered with since the digital signature also validates the message integrity as seen in the figure 23. The figure contains the following notes,

1. The message was received from identity locally called *Alice*.
2. The *encryption-key* and the *signing-key fingerprint* is displayed.
3. The note sender is trusted.
4. The note is digitally signed and the signature is valid.



Sender: Alice

Encryption-key fingerprint: e6:73:d7:b4:e2:07:ea:96:ce:cf:d3:b1:10:3c:5e:5e:db:21:67:9a

Signing-key fingerprint: d6:20:85:34:25:f3:af:dc:ed:60:65:1f:e7:0b:da:c0:31:9a:05:70

Sender is trusted. Revoke trust

Digital Signature is valid.

Note contents:

Hello Bob,

Here is today's forecast: Breezy with plenty of sun.

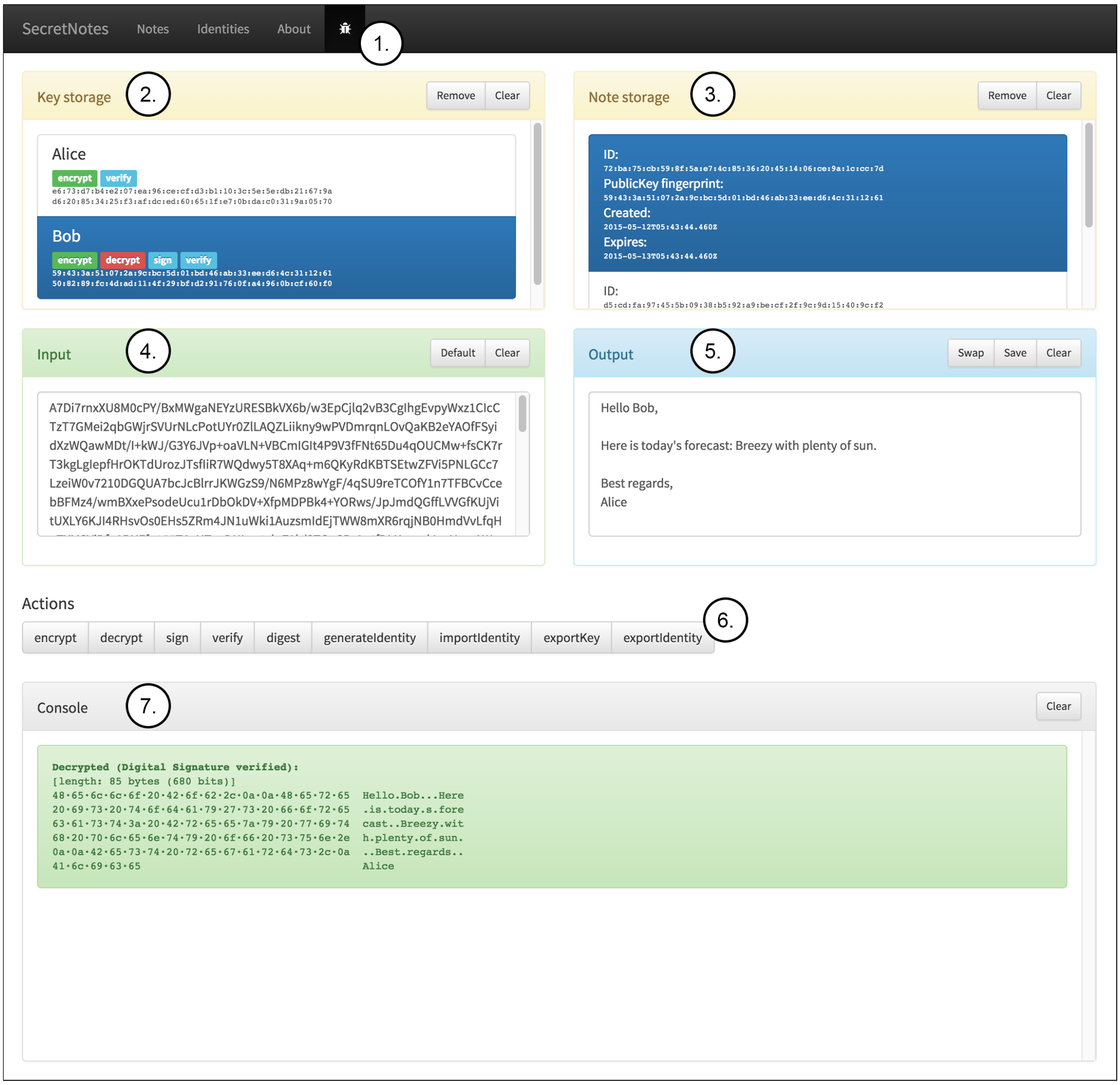
Best regards,
Alice

Figure 23: Decrypted note from trusted sender (Screenshot of the example application)

For advanced developers the example applications also contain a debug section as seen in figure 24 on the next page. The debug section can be used to inspect cryptography keys, the received notes, and perform different cryptographic operations manually. The figure contains the following notes,

1. The debug section is accessible from the main navigator.

2. The key storage contains the CryptoKey objects that have been stored in the local indexed database.
3. The note storage contains the messages that are found with the actived identity (2.).
4. The input panel displays the input data for cryptographic operation. The input panel can also be used to input data manually.
5. The output panel displays the output data from cryptograhical operation.
6. The actions contain the following operations that can be performed: encrypt, decrypt, sign, verify, digest, generateIdentity, importIdentity, exportKey and exportIdentity.
7. The console view shows the output from the operations.



Alice
e6:73:d7:b4:e2:07:ea:96:ce:cf:d3:b1:10:3c:5e:5e:db:21:67:9a
d6:20:85:34:25:f3:af:dc:ed:60:65:1f:e7:0b:da:c0:31:9a:05:70

Bob
59:43:3a:51:07:2a:9c:bc:5d:01:bd:46:ab:33:ee:d6:4c:31:12:61
50:82:89:fc:4d:ad:11:4f:29:bf:d2:91:76:0f:a4:96:0b:cf:60:f0

ID:
72:ba:75:cb:59:8f:5a:e7:4c:85:36:20:45:14:06:ce:9a:1c:cc:7d
PublicKey fingerprint:
59:43:3a:51:07:2a:9c:bc:5d:01:bd:46:ab:33:ee:d6:4c:31:12:61
Created:
2015-05-12T05:43:44.460Z
Expires:
2015-05-13T05:43:44.460Z

ID:
d5:cd:fa:97:45:5b:09:38:b5:92:a9:be:cf:2f:9c:9d:15:40:9c:f2

A7Di7rnxXU8M0cPY/BxMWgaNEYzURESBkVX6b/w3EpCjlq2vB3CgIhgEvpyWxz1CIcC
TzT7GMei2qbGWjrSVUrNLcPotUYr0ZlLAQZLiikny9wPVDmrqnLOvQaKB2eYAOfFSyi
dXzWQawMDt/I+kWJ/G3Y6JVp+oaVLN+VBCmIGIt4P9V3fFNt65Du4qOUCMw+fsCK7r
T3kgLglepfHrOKTdUrozJTsfIiR7WQdwy5T8XAq+m6QKyRdKBTSEtwZFVi5PNLGCc7
LzeiW0v7210DGQUA7bcJcBlrrJKWGzS9/N6MPz8wYgF/4qSU9reTCOfY1n7TFBCvCce
bBFMz4/wmBXxePsodeUcu1rDbOkDV+XfpMDPBk4+YORws/JpJmdQGffLVVGfKUjVi
tUXLY6KJI4RHsvOs0EHs5ZRm4JN1uWki1AuzsmIdEjTWW8mXR6rqjNB0HmdVvLfqH

Hello Bob,

Here is today's forecast: Breezy with plenty of sun.

Best regards,
Alice

```
Decrypted (Digital Signature verified):
[length: 85 bytes (680 bits)]
48•65•6c•6c•6f•20•42•6f•62•2c•0a•0a•48•65•72•65  Hello.Bob...Here
20•69•73•20•74•6f•64•61•79•27•73•20•66•6f•72•65  .is.today.s.fore
63•61•73•74•3a•20•42•72•65•65•7a•79•20•77•69•74  cast..Breezy.wit
68•20•70•6c•65•6e•74•79•20•6f•66•20•73•75•6e•2e  h.plenty.of.sun.
0a•0a•42•65•73•74•20•72•65•67•61•72•64•73•2c•0a  ..Best.regards..
41•6c•69•63•65                                   Alice
```

Figure 24: Received note from Alice now with digital signature that has been verified (Screenshot of the example application)

3.4 About the Source Code

The source code written for this project is open source software: developers are free to redistribute and/or modify it under the terms of the MIT License. License is available in appendix 3. The complete source code for the *SecretNotes* example application is available from the GitHub repository [70]. Additionally source code for the helper modules utils (utils.js) and cryptography (cryptography.js) is included in appendix 2. Since any code that uses the Web Cryptography API needs to be loaded over a HTTP Secure connection, it is necessary to setup a server to serve the files securely. A simple server that accomplishes this can be found under the server folder, and usage instructions can be found in the root folder of the GitHub repository in the README.md file. It is also possible to run the example application (with simulated server functionality that stores all notes locally in the web browser) directly [90] from GitHub which provides a web site hosting and secure connection for free.

The example application *SecretNotes* uses the following MIT licensed CSS and JavaScript libraries: Bootstrap Framework, Awesome Bootstrap Checkbox, Font Awesome, jQuery, lodash, Moment.js, Respond.js, and Mongoose.

4 Conclusion

This thesis studied how recent and upcoming improvements in browser based web cryptography and more specifically the Web Cryptography API is going to change how developers can write web applications that utilize cryptographic operations directly on the web browser. However, writing cryptographically correct applications is hard, even for cryptographically experienced developers. Great care must be taken when implementing cryptographic protocols since the Web Cryptography API provides only the building blocks, cryptographic primitives, for performing low-level operations. Incorrect protocol implementation could make a cryptographic protocol unsafe and leak information or the key material. It is possible that in the future, a high-level Web Cryptography API is provided for performing common cryptographic operations using algorithms and settings that are deemed to be safe to use. However in the end, it is the developer, who must know which algorithms are safe now and in the near future and also keep track of cryptographic developments, and keep ahead of the possible attack vectors.

The case study discussed in this thesis revealed that it is easy to write cryptographic applications that use the new Web Cryptography API specification, but since the Web Cryptography API does not provide any defaults, great care had to be taken when selecting the appropriate algorithms and values for each operation. The case study also revealed that the key management had to be implemented as part of the web application, since web browsers do not provide common interface for managing keys. However, this shortcoming makes even the most trivial web applications more complex to implement. Also to be able to move or share user’s private key material outside of the browser to another device, platform or web browser the keys must be created as exportable which makes keys vulnerable for attacks targeted to the JavaScript environment.

Finally, the investigation revealed that Web Cryptography API specification is an important addition to the web browser JavaScript capabilities, but in order for it to be successful in enabling creation of cryptographically secure web applications, the JavaScript environment needs a support from other security related technologies like the Content Security

Policy and Subresource Integrity specification. Also key management in the Web Cryptography API is currently limited to the browser environment and new technologies like the upcoming WebCrypto Key Discovery specification and other related specifications for managing key material outside browser environment should be investigated.

References

1 Ecma International. ECMAScript Language Specification [online]. Ecma International; 2011. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm. Accessed April 21, 2015.

2 AOL. Netscape [online]. AOL; 2015. URL: http://netscape.aol.com. Accessed January 04, 2015.

3 Oracle Corporation. Oracle and Sun Microsystems [online]. Oracle Corporation; 2015. URL: http://www.oracle.com/us/sun/index.html. Accessed January 04, 2015.

4 Oracle Corporation. Java Language and Virtual Machine Specifications [online]. Oracle Corporation; 2015. URL: https://docs.oracle.com/javase/specs/. Accessed April 17, 2015.

5 Severance C. JavaScript: Designing a Language in 10 Days [serial online]. Computer. 2012 feb;45(2):7–8. URL: http://dx.doi.org/10.1109/mc.2012.57.

6 Gartner, Inc . Technology Research [online]. Gartner, Inc.; 2015. URL: http://www.gartner.com. Accessed January 04, 2015.

7 Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika D Navara, Edward O'Connor, et al.. HTML5 [online]. W3C; 2014. URL: http://www.w3.org/TR/html5/. Accessed February 22, 2015.

8 Gartner, Inc . Gartner Identifies the Top 10 Strategic Technology Trends for 2014 [online]. Gartner, Inc.; 2014. URL: http://www.gartner.com/newsroom/id/2603623. Accessed April 05, 2015.

9 Node js Foundation. Node.js [online]. Node.js Foundation; 2015. URL: https://nodejs.org. Accessed January 04, 2015.

10 Node js Foundation. JavaScript I/O - io.js [online]. Node.js Foundation; 2015. URL: https://iojs.org. Accessed January 04, 2015.

11 Forbes, Inc . Home Depot Credit Card Breach Could Prove To Be Larger Than Target Breach [online]. Forbes, Inc.; 2014. URL: http://www.forbes.com/sites/katevinton/2014/09/03/data-breach-bulletin-home-depot-credit-card-breach-could-prove-to-be-larger-than-target-breach/. Accessed April 05, 2015.

12 Brian Krebs. Krebs On Security: Data Breaches Post Category [online]. Krebs on Security; 2015. URL: http://krebsonsecurity.com/category/data-breaches/. Accessed March 20, 2015.

13 Ars Technica. In The Wild: Phony SSL Certificates Impersonating Google, Facebook, and iTunes [online]. Condé Nast; 2014. URL: http://arstechnica.com/security/2014/02/in-the-wild-phony-ssl-certificates-impersonating-google-facebook-and-itunes/. Accessed March 20, 2015.

14 Courage Foundation. Surveillance Programs [online]. Courage Foundation; 2015. URL: https://www.freesnowden.is/surveillance-programs. Accessed March 20, 2015.

15 National Security Agency. Welcome to the National Security Agency [online]. National Security Agency; 2015. URL: https://www.nsa.gov/. Accessed January 04, 2015.

16 Landau S. Making Sense from Snowden: What's Significant in the NSA Surveillance Revelations. Security Privacy, IEEE. 2013 July;11(4):54–63.

17 Pew Research Center. About Pew Research Center [online]. 2015;URL: http://www.pewresearch.org/about/. Accessed April 05, 2015.

18 Pew Research Center. Public Perceptions of Privacy and Security in the Post-Snowden Era [online]. 2014;URL: http://www.pewinternet.org/files/2014/11/PI_PublicPerceptionsofPrivacy_111214.pdf. Accessed March 15, 2015.

19 Symantec Corporation. Symantec Intelligence Report, January 2015 [online]. Symantec Corporation; 2015. URL: http://www.symantec.com/content/en/us/enterprise/other_resources/b-intelligence-report-01-2015-en-us.pdf. Accessed April 15, 2015.

20 Symantec Corporation. 2015 Internet Security Threat Report, Volume 20 [online]. Symantec Corporation; 2015. URL: http://www.symantec.com/security_response/publications/threatreport.jsp. Accessed April 15, 2015.

21 Symantec Corporation. Press release: Deceptive New Tactics Give Advanced Attackers Free Reign Over Corporate Networks [online]. Symantec Corporation; 2015. URL: http://www.symantec.com/about/news/release/article.jsp?prid=20150414_01. Accessed May 15, 2015.

22 Halpin H. The W3C Web Cryptography API: Motivation and Overview [serial online]. In: Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion. WWW Companion '14. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee; 2014. p. 959–964. URL: http://dx.doi.org/10.1145/2567948.2579224.

23 Stanford University. Stanford Javascript Crypto Library [online]. Stanford University; 2009. URL: http://bitwiseshiftleft.github.io/sjcl/. Accessed March 02, 2015.

24 Jeff Mott. CryptoJS: JavaScript Implementations of Standard and Secure Cryptographic Algorithms [online]. Jeff Mott; 2013. URL: https://code.google.com/p/crypto-js/. Accessed April 10, 2015.

25 Forge: A Native Implementation of TLS (And Various Other Cryptographic Tools) in JavaScript [online]. Digital Bazaar; 2015. URL: https://github.com/digitalbazaar/forge/. Accessed April 10, 2015.

26 Google, Inc . V8 JavaScript Engine [online]. Google, Inc.; 2015. URL: https://code.google.com/p/v8/. Accessed April 17, 2015.

27 Matasano Security. Javascript Cryptography Considered Harmful [online]. Matasano Security; 2014. URL: http://matasano.com/articles/javascript-cryptography. Accessed March 01, 2015.

28 Lawson N. Side-Channel Attacks on Cryptographic Software [serial online]. IEEE Security & Privacy Magazine. 2009 nov;7(6):65–68. URL: http://dx.doi.org/10.1109/msp.2009.165.

29 W3C. W3C Web Cryptography Working Group [online]. W3C; 2014. URL: http://www.w3.org/2012/webcrypto/. Accessed January 04, 2015.

30 W3C. World Wide Web Consortium Process Document [online]. W3C; 2014. URL: http://www.w3.org/2014/Process-20140801/. Accessed April 15, 2015.

31 W3C. Web Cryptography API [online]. W3C; 2014. URL: http://www.w3.org/TR/2014/CR-WebCryptoAPI-20141211/. Accessed February 22, 2015.

32 Rescorla E. HTTP Over TLS [online]. RFC Editor; 2000. 2818. URL: http://www.rfc-editor.org/rfc/rfc2818.txt.

33 W3C. About W3C [online]. W3C; 2015. URL: http://www.w3.org/Consortium/. Accessed January 04, 2015.

34 Mozilla Foundation. Web Workers API [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API. Accessed April 17, 2015.

35 Arun Ranganathan. Web Cryptography API Use Cases [online]. W3C; 2013. URL: http://www.w3.org/TR/webcrypto-usecases/. Accessed February 28, 2015.

36 Ian Goldberg. Off-The-Record Messaging Protocol Version 3 [online]. Cypherpunks Canada; 2008. URL: https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html. Accessed February 22, 2015.

37 Scott Helme. Perfect Forward Secrecy [online]. Scott Helme; 2014. URL: https://scotthelme.co.uk/perfect-forward-secrecy/. Accessed February 18, 2015.

38 Stark E, Hamburg M, Boneh D. Symmetric Cryptography in Javascript [serial online]. 2009 dec;URL: http://dx.doi.org/10.1109/acsac.2009.42.

39 Mozilla Foundation. DOMCryptAPISpec [online]. Mozilla Foundation; 2011. URL: https://wiki.mozilla.org/Privacy/Features/DOMCryptAPISpec/Latest. Accessed April 17, 2015.

40 Folsom, W Davis. Understanding American Business Jargon: A Dictionary. Greenwood Publishing Group; 2005.

41 W3C. Security Activity Statement [online]. W3C; 2015. URL: http://www.w3.org/Security/Activity. Accessed April 17, 2015.

42 W3C. Web Cryptography Working Group Charter [online]. W3C; 2014. URL: http://www.w3.org/2011/11/webcryptography-charter.html. Accessed January 04, 2015.

43 W3C. Identity in the Browser Workshop [online]. W3C; 2011. URL: http://www.w3.org/2011/identity-ws/report.html. Accessed January 15, 2015.

44 W3C. Suomen Toimisto [online]. W3C; 2015. URL: http://www.w3c.tut.fi/. Accessed Januarytho 04, 2015.

45 IEEE Standard Specifications for Public-Key Cryptography. IEEE Std 1363-2000. 2000 Aug;p. 1–228.

46 Garfinkel, Simson. PGP: Pretty Good Privacy. 1st ed. Russell, Deborah, editor. Sebastopol, CA, USA: O'Reilly & Associates, Inc.; 1996.

47 CGI, Inc . Public Key Encryption and Digital Signature: How Do They Work? [online]. CGI, Inc.; 2004. URL: http://www.cgi.com/files/white-papers/cgi_whpr_35_pki_e.pdf. Accessed February 25, 2015.

48 European Parliament. Directive 1999/93/EC [online]. Brussels: EUR-Lex; 1999. URL: http://eur-lex.europa.eu/legal-content/EN/LSU/?uri=CELEX:31999L0093. Accessed May 05, 2015.

49 Rivest RL. The MD5 Message-Digest Algorithm [online]. RFC Editor; 1992. 1321. URL: http://www.rfc-editor.org/rfc/rfc1321.txt.

50 FIPS PUB 180-4: Secure Hash Standard (SHS) [online]. Gaithersburg, MD, United States; 2012. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf.

51 Mozilla Foundation. Crypto Interface [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/Crypto. Accessed February 23, 2015.

52 Mozilla Foundation. Window [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window. Accessed February 23, 2015.

53 Mozilla Foundation. Worker [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/Worker. Accessed February 23, 2015.

54 Mozilla Foundation. SubtleCrypto Interface [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto. Accessed February 23, 2015.

55 Mozilla Foundation. GetRandomValues Function [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/RandomSource/getRandomValues. Accessed February 23, 2015.

56 Eastlake D, Schiller J, Crocker S. Randomness Requirements for Security [online]. RFC Editor; 2005. 106. URL: http://www.rfc-editor.org/rfc/rfc4086.txt.

57 Jonsson J, Kaliski B. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 [online]. RFC Editor; 2003. 3447. URL: http://www.rfc-editor.org/rfc/rfc3447.txt.

58 Rivest RL, Shamir A, Adleman L. A Method for Obtaining Digital Signatures and Public-key Cryptosystems [serial online]. Commun ACM. 1978 Feb;21(2):120–126. URL: http://doi.acm.org/10.1145/359340.359342.

59 RSA Laboratories. RSA Algorithm [online]. EMC Corporation; 2015. URL: http://www.emc.com/emc-plus/rsa-labs/historical/rsa-algorithm.htm. Accessed April 10, 2015.

60 Dworkin MJ. SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation: Methods and Techniques [online]. Gaithersburg, MD, United States; 2001. URL: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf.

61 Dworkin MJ. SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC [online]. Gaithersburg, MD, United States; 2007. URL: http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf.

62 ANSI A. X9. 62: 2005: Public Key Cryptography for the Financial Services Industry. The elliptic curve digital signature algorithm (ECDSA). 2005;.

63 Dworkin MJ. SP 800-38B. Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication [online]. Gaithersburg, MD, United States; 2005. URL: http://csrc.nist.gov/publications/nistpubs/800-38B/SP_800-38B.pdf.

64 FIPS PUB 198-1: The Keyed-Hash Message Authentication Code (HMAC) [online]. Gaithersburg, MD, United States; 2008. URL: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf.

65 Mozilla Foundation. Promise Object [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Accessed April 17, 2015.

66 Colin Toh. Staying Sane With Asynchronous Programming: Promises and Generators [online]. Colin Toh; 2014. URL: http://colintoh.com/blog/staying-sane-with-asynchronous-programming-promises-and-generators. Accessed April 17, 2015.

67 Pyramid of Doom [online]. SurviveJS; 2014. URL: http://survivejs.com/common_problems/pyramid.html. Accessed April 17, 2015.

68 Mozilla Foundation. ECMAScript 6 Draft [online]. Mozilla Foundation; 2015. URL: http://wiki.ecmascript.org/doku.php?id=harmony: specification_drafts#draft_specification_for_es.next_ecma-262_edition_6. Accessed April 17„ 2015.

69 Mozilla Foundation. Error Object [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error. Accessed April 17, 2015.

70 Mika Luoma-aho. SecretNote Source Code in GitHub Repository [online]; 2015. URL: https://github.com/webcryptoapiex/secretnote. Accessed April 20, 2015.

71 Jones M. JSON Web Key (JWK) [online]. IETF Secretariat; 2015. draft-ietf-jose-json-web-key-41. URL: http://www.ietf.org/internet-drafts/draft-ietf-jose-json-web-key-41.txt.

72 W3C. Indexed Database API [online]. W3C; 2015. URL: http://www.w3.org/TR/IndexedDB/. Accessed February 22, 2015.

73 Alexis Deveria. Browser Support for WebCryptoAPI [online]. Alexis Deveria; 2015. URL: http://caniuse.com/#feat=cryptography. Accessed April 08, 2015.

74 Remy Sharp. About [online]; 2015. URL: https://remysharp.com/about/. Accessed January 04, 2015.

75 Remy Sharp. What is a Polyfill [online]. Remy Sharp; 2010. URL: https://remysharp.com/2010/10/08/what-is-a-polyfill/. Accessed February 25, 2015.

76 Raytheon BBN Technologies Corp . PolyCrypt: Web Cryptography API Polyfill [online]. Raytheon BBN Technologies Corp.; 2015. URL: http://polycrypt.net. Accessed February 25, 2015.

77 Netflix, Inc . NfWebCrypto: Web Cryptography API Polyfill [online]. Netflix, Inc.; 2014. URL: https://github.com/Netflix/nfwebcrypto/. Accessed April 17, 2015.

78 Mark Watson. WebCrypto Key Discovery API [online]. W3C; 2014. URL: http://dvcs.w3.org/hg/webcrypto-keydiscovery/raw-file/tip/Overview.html. Accessed February 22, 2015.

79 Mozilla Foundation. Same-Origin Policy [online]. Mozilla Foundation; 2015. URL: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed February 23, 2015.

80 Barth A. The Web Origin Concept [online]. RFC Editor; 2011. 6454. URL: http://www.rfc-editor.org/rfc/rfc6454.txt.

81 W3C. Content Security Policy Level 2 [online]. W3C; 2015. URL: http://www.w3.org/TR/CSP2/. Accessed January 18, 2015.

82 Evans C, Palmer C, Sleevi R. Public Key Pinning Extension for HTTP [online]. RFC Editor; 2015. 7469. URL: http://www.rfc-editor.org/rfc/rfc7469.txt.

83 OWASP Foundation. Man-In-The-Middle Attack [online]. OWASP Foundation; 2014. URL: https://www.owasp.org/index.php/Man-in-the-middle_attack. Accessed April 20, 2015.

84 W3C. Subresource Integrity [online]. W3C; 2015. URL: http://www.w3.org/TR/SRI/. Accessed April 20, 2015.

85 Arce I. The Weakest Link Revisited [serial online]. IEEE Security & Privacy Magazine. 2003 mar;1(2):72–76. URL: http://dx.doi.org/10.1109/msecp.2003.1193216.

86 CGISecurity. The Cross-Site Scripting (XSS) FAQ [online]. CGISecurity; 2015. URL: http://www.cgisecurity.com/xss-faq.html. Accessed May 09, 2015.

87 Lee M. Encryption Works: How to Protect Your Privacy in the Age of NSA Surveillance [online]. 2013 jul;URL: https://freedom.press/encryption-works.

88 Eric Rescorla. Diffie-Hellman Key Agreement Method [online]. RFC Editor; 1999. 2631. URL: http://www.rfc-editor.org/rfc/rfc2631.txt.

89 Diffie, W , Hellman, M E . New Directions in Cryptography. Information Theory, IEEE Transactions on. 1976 Nov;22(6):644–654.

90 Mika Luoma-aho. SecretNote Live Demo (Standalone Version) [online]; 2015. URL: http://webcryptoapiex.github.io/secretnote/. Accessed April 20, 2015.

1 List of Algorithms

Algorithm name	encrypt	decrypt	sign	verify	digest	generate key	derive key	derive bits	import key	export key	wrap key	unwrap key
RSASSA-PKCS1-v1_5			✔	✔		✔			✔	✔		
RSA-PSS			✔	✔		✔			✔	✔		
RSA-OAEP	✔	✔				✔			✔	✔	✔	✔
ECDSA			✔	✔		✔			✔	✔		
ECDH						✔	✔	✔	✔	✔		
AES-CTR	✔	✔				✔			✔	✔	✔	✔
AES-CBC	✔	✔				✔			✔	✔	✔	✔
AES-CMAC			✔	✔		✔			✔	✔		
AES-GCM	✔	✔				✔			✔	✔	✔	✔
AES-CFB	✔	✔				✔			✔	✔	✔	✔
AES-KW						✔			✔	✔	✔	✔
HMAC			✔	✔		✔			✔	✔		
DH						✔	✔	✔	✔	✔		
SHA-1					✔							
SHA-256					✔							
SHA-384					✔							
SHA-512					✔							
CONCAT							✔	✔	✔			
HKDF-CTR							✔	✔	✔			
PBKDF2						✔	✔	✔	✔			

2 Source Code

The complete source code for the example application, example server implementation and the helper modules can be found in the GitHub repository [70]. The source code for the utilities and cryptography helper modules can also be found in this appendix.

Listing 14 includes the complete JavaScript source code for the utilities helper module (utils.js).

```javascript
// Copyright 2015 Mika "Fincodr" Luoma-aho
// Provided under the MIT license. See LICENSE file for details.
(function(parent){
  "use strict";

  // The main application module
  var app = parent.app = parent.app || {};

  // utils module
  app.utils = (function(){

    var self = this;

    var module = {

      stringPadRight: function(str, len, ch) {
        var chx = ch || ' ';
        while(str.length < len) {
          str += chx;
        }
        return str;
      },

      stringPadLeft: function(s, len, ch) {
        var str = '', chx = ch || ' ';
        while(str.length + s.length < len) {
          str += chx;
        }
        str += s;
        return str;
      },

      compareTwoUint8Arrays: function(a,b) {
        if (a.length===b.length) {
          for (var i=0, len=a.length; i!==len; ++i) {
            if (a[i]!==b[i]) {
              return false;
            }
          }
          return true;
        }
        return false;
      },

      convertTextToUint8Array: function(s) {
        var data = new Uint8Array(s.length);
        for (var i=0, len=s.length; i!==len; ++i) {
          data[i] = s.charCodeAt(i);
        }
        return data;
      },

      convertTextToArrayBuffer: function(s) {
        var buf = new ArrayBuffer(s.length);
        var view = new Uint8Array(buf);
        for (var i=0, len=s.length; i!==len; ++i) {
          view[i] = s.charCodeAt(i);
        }
        return buf;
      },

      packUint8Arrays: function() {
        // generate big enough new array z
        var i, len, ptr, count = arguments.length,
           totalLength = 0;
        for (i=0; i!==count; ++i) {
          if (arguments[i]) {
            totalLength += arguments[i].length;
          }
        }
        var z = new Uint8Array(totalLength + count*2);
        // copy data
        for (i=0, ptr=0; i!==count; ++i) {
          if (arguments[i]) {
            len = arguments[i].length;
            z.set(arguments[i], ptr + 2);
          } else {
            len = 0;
          }
          var datalen = new Uint16Array(2);
          datalen[0] = len >> 8;
          datalen[1] = len - (datalen[0]*256);
          if (len > 65535) {
            throw new Error('packUint8Arrays supports max length of 65535 bytes of data per packed component');
          }
          z.set(datalen, ptr);
          ptr += len + 2;
        }
        return z;
      },

      unpackUint8Arrays: function(data) {
        var i = 0, len, ptr = 0, totalLength = data.length;
        var z = [];
        // copy data
        while (ptr < totalLength) {
          len = data[ptr] * 256 + data[ptr+1];
          if (ptr+2+len > totalLength) {
            throw new Error('unpackUint8Arrays out of bounds!');
          }
          z.push(data.subarray(ptr + 2, ptr + 2 + len));
          ptr += len + 2;
          ++i;
        }
        return z;
      },

      concatUint8Arrays: function() {
        // generate big enough new array z
        var i, ptr = 0, totalLength = 0;
        for (i=0; i!==arguments.length; ++i) {
          totalLength += arguments[i].length;
        }
        var z = new Uint8Array(totalLength);
        // copy data
        for (i=0; i!==arguments.length; ++i) {
          z.set(arguments[i], ptr);
          ptr += arguments[i].length;
        }
        return z;
      },

      convertBase64ToUint8Array: function(data) {
        var binary = window.atob(data);
        var len = binary.length;
        var buf = new ArrayBuffer(len);
        var view = new Uint8Array(buf);
        for (var i=0; i!==len; ++i) {
          view[i] = binary.charCodeAt(i);
        }
        return view;
      },

      convertUint8ArrayToBase64: function(data) {
        var s = module.convertUint8ArrayToText(data);
        return window.btoa(s);
      },

      convertUint8ArrayToText: function(data) {
        var s = '';
        for (var i=0, len=data.length; i!==len; ++i) {
          s += String.fromCharCode(data[i]);
        }
        return s;
      },

      convertArrayBufferToText: function(data) {
        var s = '';
        for (var i=0, len=data.byteLength; i!==len; ++i) {
          s += String.fromCharCode(data[i]);
        }
        return s;
      },

      convertArrayBufferToUint8Array: function(data) {
        var a = new Uint8Array(data.byteLength);
        for (var i=0, len=data.byteLength; i!==len; ++i) {
          a[i] = data[i];
        }
        return a;
      },

      convertUint8ArrayToArrayBuffer: function(data) {
        var a = new ArrayBuffer(data.length);
        for (var i=0, len=data.length; i!==len; ++i) {
          a[i] = data[i];
        }
        return a;
      },

      convertUint8ArrayToHex: function(data, sep) {
        var a, h = '';
        var ch = sep?sep:'';
        for (var i=0, len=data.length; i!==len; ++i) {
          a = data[i];
          h += i>0?ch:'';
          h += a<16?'0':'';
          h += a.toString(16);
        }
```

```
        return h;
      },

      convertHexToUint8Array: function(data) {
        var len = data.length;
        var a = new Uint8Array(len/2);
        for (var i=0, j=0; i!==len; i+=2) {
          a[j++] = parseInt(data.substr(i, 2), 16);
        }
        return a;
      },

      convertUint8ArrayToHexView: function(data, width, sep) {
        var a, h = '', s = '';
        var ch = sep===undefined?' ':sep;
        var n = 0;
        h = '[length: ' + data.length + ' bytes (' + data.length * 8 + ' bits)]\n';
        for (var i=0, len=data.length; i!==len; ++i) {
          a = data[i];
          h += n>0?ch:'';
          h += a<16?'0':'';
          h += a.toString(16);
          n++;
          s += ((a>=97 && a<=122)|(a>=65 && a<=90)|(a>48 && a<=57))?String.fromCharCode(a):'.';
          if (n===width) {
            h += '  ' + s;
            h += '\n';
            n=0;
            s='';
          }
        }
        if (n!==0) {
          h += '  ' + module.stringPadLeft('', (width-n)*3) + s;
        }
        return h;
      }

    };

    return module;

  })();

})(this); // this = window
```

Listing 14: Utility JavaScript module (utils.js)

Listing 15 includes the complete JavaScript source code for the cryptography helper module (cryptography.js).

```
// Copyright 2015 Mika "Fincodr" Luoma-aho
// Provided under the MIT license. See LICENSE file for details.
(function(parent){
  "use strict";

  // The main application module
  var app = parent.app = parent.app || {};

  // fix safari crypto namespace
  //
  if (window.crypto && !window.crypto.subtle && window.crypto.webkitSubtle) {
    window.crypto.subtle = window.crypto.webkitSubtle;
  }

  /**
   * Detect Web Cryptography API
   * @return {Boolean} true, if success
   */
  function isWebCryptoAPISupported() {
    return 'crypto' in window && 'subtle' in window.crypto;
  }

  // crypto module
  app.cryptography = (function(){

    var self = this;

    var module = {

      isSupported: function() {
        // check that we have Crypto interface
        if ("crypto" in window) {
          // check that we have SubtleCryto interface
          if ("subtle" in window.crypto) {
            return true;
          }
        }
        return false;
      },

      returnResolve: function(value) {
        return new Promise(function(resolve, reject) {
          resolve(value);
        });
      },

    digest: function(alg, data) {
      return window.crypto.subtle.digest(
        alg,
        data
      );
    },

    generateKeys: function(alg, exportable, usage) {
      return window.crypto.subtle.generateKey(
        alg, // algorithm
        exportable, // non-exportable
        usage // usage
      );
    },

    importKey: function(key, alg, format, exportable,
     usage) {
      return window.crypto.subtle.importKey(
        format, // raw format
        key, // key to import
        alg, // algorithm
        exportable, // exportable
        usage // key usages
      );
    },

    exportKey: function(key, format) {
      return window.crypto.subtle.exportKey(
        format, // raw format
        key // key to export
      );
    },

    encryptData: function(alg, key, inputData) {
      return window.crypto.subtle.encrypt(
        alg, // algorithm
        key, // key to use for encryption
        inputData // input data
      );
    },

    decryptData: function(alg, key, inputData) {
      return window.crypto.subtle.decrypt(
        alg, // algorithm
        key, // key to use for decryption
        inputData // input data
      );
    },

    signData: function(alg, key, inputData) {
      return window.crypto.subtle.sign(
        alg, // algorithm
        key, // key to use for signing
        inputData // input data
      );
    },

    exportIdentity: function(publicKey, privateKey, signingKey, verifyKey) {
      return new Promise(function(resolve, reject) {
        var exported = {};

        function exportKeyOrContinue(key, format, c, k) {
          return new Promise(function(done, fail) {
            // Try to export publicKey
            module.exportKey(key, format)
            .then(function(result) {
              c[k] = new Uint8Array(result);
              done();
            })
            .catch(function(err) { done(); });
          });
        }

        exportKeyOrContinue(publicKey, 'spki', exported, 'publicKeyData')
        .then(function(result) {
          return exportKeyOrContinue(privateKey, 'pkcs8', exported, 'privateKeyData');
        })
        .then(function(result) {
          return exportKeyOrContinue(verifyKey, 'spki', exported, 'verifyKeyData');
        })
        .then(function(result) {
          return exportKeyOrContinue(signingKey, 'pkcs8', exported, 'signingKeyData');
        })
        .then(function(){
          // Concat arrays
          if (!exported.verifyKeyData) {
            exported.publicIdentityData = app.utils.packUint8Arrays(
              exported.publicKeyData
            );
          } else {
            exported.publicIdentityData = app.utils.packUint8Arrays(
              exported.publicKeyData,
              exported.verifyKeyData
            );
          }
          if (!exported.signingKeyData) {
            exported.privateIdentityData = app.utils.packUint8Arrays(
              exported.privateKeyData
            );
          } else {
            exported.privateIdentityData = app.utils.packUint8Arrays(
              exported.privateKeyData,
              exported.signingKeyData
            );
          }
          resolve(exported);
        })
        .catch(function(err){
          reject(err);
        });
      });
    },

    importIdentity: function(asymAlg, signingAlg, publicIdentityData, privateIdentityData, exportablePrivateIdentity) {
      return new Promise(function(resolve, reject) {
        var imported = {};

        function importKeyOrContinue(key, alg, format, exportable, usage) {
          return new Promise(function(done, fail) {
            if (key) {
              // Try to import key
              module.importKey(key, alg, format, exportable, usage)
              .then(function(result) { done(result); })
              .catch(function(err) { done(); });
            } else {
              // Fail but continue
              done();
            }
          });
        }

        var publicIdentity = app.utils.unpackUint8Arrays(publicIdentityData);
        var privateIdentity = app.utils.unpackUint8Arrays(privateIdentityData);

        imported = {
          publicKeyData: publicIdentity[0],
          verifyKeyData: publicIdentity[1],
          privateKeyData: privateIdentity[0],
          signingKeyData: privateIdentity[1]
        };

        importKeyOrContinue(imported.publicKeyData, asymAlg, 'spki', true, ['encrypt'])
        .then(function(result) {
          imported.publicKey = result;
          return importKeyOrContinue(imported.verifyKeyData, signingAlg, 'spki', true, ['verify']);
        })
        .then(function(result) {
          imported.verifyKey = result;
          return importKeyOrContinue(imported.privateKeyData, asymAlg, 'pkcs8', exportablePrivateIdentity, ['decrypt']);
        })
        .then(function(result) {
          imported.privateKey = result;
          return importKeyOrContinue(imported.signingKeyData, signingAlg, 'pkcs8', exportablePrivateIdentity, ['sign']);
        })
        .then(function(result) {
          imported.signingKey = result;
          resolve(imported);
        });
      });
    },

    verifyData: function(alg, key, digitalSignature, inputData) {
      return window.crypto.subtle.verify(
        alg, // algorithm
        key, // key to use for signing
        digitalSignature,
        inputData // input data
      );
    },

    encryptAndSign: function(asymAlg, symAlg, signingAlg, plaintext, encryptionKey, signingKey, verifyKey, publicKey) {
      return new Promise(function(resolve, reject) {

        var state = {};
```

```
state.plaintextUint8Array = app.utils.convertTextToUint8Array(plaintext);

state.signed = signingKey?true:false;
state.hasSignature = state.signed?true:false;
state.hasPublicKey = publicKey?true:false;

// generate IV for symmetric encryption (symmetricIV)
state.symmetricIV = window.crypto.getRandomValues(new Uint8Array(16));

// generate symmetric key (symmetricKey)
module.generateKeys(symAlg, true, ['encrypt', 'decrypt'])
.then(function(symmetricKey) {
  state.symmetricKey = symmetricKey;
  // export generated key
  return module.exportKey(state.symmetricKey, 'raw');
})
.then(function(exportedSymmetricKey) {
  state.exportedSymmetricKey = new Uint8Array(exportedSymmetricKey);
  if (state.signed && verifyKey) {
    // if verify-key is provided, export the key
    return module.exportKey(verifyKey, 'spki');
  } else {
    // if verify-key is not provided, continue to next step
    return module.returnResolve(false);
  }
})
.then(function(exportedVerifyKey) {
  if (state.signed && verifyKey) {
    state.exportedVerifyKey = new Uint8Array(exportedVerifyKey);
    // if signing-key was provided, sign the plaintext
    return module.signData(signingAlg, signingKey, state.plaintextUint8Array);
  } else {
    // if signing-key was not provided, continue to next step
    state.exportedVerifyKey = new Uint8Array();
    return module.returnResolve();
  }
})
.then(function(digitalSignature) {
  state.digitalSignature = new Uint8Array(
   digitalSignature);
 if (state.hasPublicKey) {
  // if public-key was provided, export the key
  return module.exportKey(publicKey, 'spki');
 } else {
  // if public-key was not provided, continue to next step
  return module.returnResolve();
 }
})
.then(function(exportedPublicKey) {
 if (state.hasPublicKey) {
  state.exportedPublicKey = new Uint8Array(exportedPublicKey);
 }
 if (state.signed) {
  // if signing, create package from: [ plaintext, digitalsignature, verifyKey, (optional)publicKey ]
  state.dataToEncrypt = app.utils.packUint8Arrays(
   new Uint8Array([1, state.hasPublicKey?1:0]),
   state.plaintextUint8Array,
   state.digitalSignature,
   state.exportedVerifyKey,
   state.exportedPublicKey
  );
 } else {
  // if not signing, create package from: [ plaintext, (optional)publicKey ]
  state.dataToEncrypt = app.utils.packUint8Arrays(
   new Uint8Array([0, state.hasPublicKey?1:0]),
   state.plaintextUint8Array,
   state.exportedPublicKey
  );
 }
 symAlg.iv = state.symmetricIV;
 // encrypt the package that was created on previous step
 return module.encryptData(symAlg, state.symmetricKey, state.dataToEncrypt);
})
.then(function(encryptedDataArray) {
 state.encryptedPlaintextAndDigitalSignatureAndVerifyKey = new Uint8Array(encryptedDataArray);
 // create package from: [ symmetricKey, symmetricIV ]
 state.symmetricKeyAndIVpack = app.utils.
```

```
              packUint8Arrays(
            state.exportedSymmetricKey,
            state.symmetricIV
          );
          // encrypt the package that was created on the previous step
          return module.encryptData(encryptionKey.algorithm, encryptionKey, state.symmetricKeyAndIVpack);
        })
        .then(function(encryptedSymmetricKeyAndIVArray) {
          state.encryptedSymmetricKeyAndIV = new Uint8Array(encryptedSymmetricKeyAndIVArray);
          // create output package from: [ [plaintext+((optional)digitalsignature+verifykey)+(optionally)publickey] + [symK+symIV] ]
          state.packedCipher = app.utils.packUint8Arrays(
            state.encryptedPlaintextAndDigitalSignatureAndVerifyKey,
            state.encryptedSymmetricKeyAndIV
          );
          resolve(state);
        })
        .catch(function(err) {
          // if rejected in any point of the process, report the error
          reject(err);
        });
      });
    },

    decryptAndVerify: function(asymAlg, symAlg, signingAlg, digestAlg, packedCipher, decryptionKey) {
      return new Promise(function(resolve, reject) {
        var state = {
          signed: false
        };

        var unpackedCipher = app.utils.unpackUint8Arrays(packedCipher);

        state.encryptedPlaintextAndDigitalSignatureAndVerifyKey = unpackedCipher[0];
        state.encryptedSymmetricKeyAndIV = unpackedCipher[1];

        module.decryptData(decryptionKey.algorithm, decryptionKey, state.encryptedSymmetricKeyAndIV
           )
        .then(function(result) {
          var symmetricKeyAndIV = app.utils.
             unpackUint8Arrays(new Uint8Array(result));
          state.symmetricKeyData = symmetricKeyAndIV[0];
          state.symmetricIV = symmetricKeyAndIV[1];
          return module.importKey(state.symmetricKeyData,
             symAlg, 'raw', false, ['decrypt']);
        })
        .then(function(result) {
          state.symmetricKey = result;
          symAlg.iv = state.symmetricIV;
          return module.decryptData(symAlg, state.
             symmetricKey, state.
             encryptedPlaintextAndDigitalSignatureAndVerifyKey
             );
        })
        .then(function(result) {
          var plaintextAndDigitalSignatureAndVerifyKey =
             app.utils.unpackUint8Arrays(new Uint8Array(
             result));
          state.hasSignature =
             plaintextAndDigitalSignatureAndVerifyKey
             [0][0];
          state.hasPublicKey =
             plaintextAndDigitalSignatureAndVerifyKey
             [0][1];
          state.plaintextUint8Array =
             plaintextAndDigitalSignatureAndVerifyKey[1];
          if (state.hasSignature) {
            // Is digitally signed
            state.signed = true;
            state.digitalSignature =
               plaintextAndDigitalSignatureAndVerifyKey
               [2];
            state.verifyKeyData =
               plaintextAndDigitalSignatureAndVerifyKey
               [3];
            state.publicKeyData = state.hasPublicKey?
               plaintextAndDigitalSignatureAndVerifyKey
               [4]:undefined;
            return module.digest(digestAlg, state.
               verifyKeyData);
          } else {
            // Not digitally signed
            state.publicKeyData = state.hasPublicKey?
               plaintextAndDigitalSignatureAndVerifyKey
               [2]:undefined;
            return module.returnResolve();
          }
        })
        .then(function(hash) {
          if (state.signed) {
            state.verifyKeyFingerprint = new Uint8Array(hash);
            return module.importKey(state.verifyKeyData, signingAlg, 'spki', true, ['verify']);
          } else {
            return module.returnResolve();
          }
        })
        .then(function(result) {
          if (state.signed) {
            state.verifyKey = result;
          }
          if (state.publicKeyData) {
            return module.importKey(state.publicKeyData, asymAlg, 'spki', true, ['encrypt']);
          } else {
            return module.returnResolve();
          }
        })
        .then(function(publicKey) {
          state.publicKey = publicKey;
          if (state.publicKeyData) {
            return module.digest(digestAlg, state.publicKeyData);
          } else {
            return module.returnResolve();
          }
        })
        .then(function(hash) {
          if (state.publicKeyData) {
            state.publicKeyFingerprint = new Uint8Array(hash);
          }
          if (state.signed) {
            return module.verifyData(signingAlg, state.verifyKey, state.digitalSignature, state.plaintextUint8Array);
          } else {
            return module.returnResolve(false);
          }
        })
        .then(function(result) {
          state.digitalSignatureValid = result;
          resolve(state);
        })
        .catch(function(err){
          reject(err);
```

```
                });
            });
        }

    };

    return module;

  })();

})(this); // this = window
```

Listing 15: Cryptography JavaScript module (cryptography.js)

3 MIT License

The MIT License (MIT)