# A Javascript voting client for remote online voting

Jordi Cucurull[1] and Sandra Guasch[1] and David Galindo[2]

[1] Scytl Secure Online Voting, Pl. Gal·la Placídia, 1-3, 1st floor, 08006 Barcelona - Spain
{jordi.cucurull, sandra.guasch}@scytl.com
[2] School of Computer Science, The University of Birmingham, Edgbaston, Birmingham, B15 2TT, United Kingdom
d.galindo@cs.bham.ac.uk

**Abstract.** Remote electronic voting systems enable elections where voters can vote remotely without geographical constraints using their own devices, e.g. smartphones, PCs or other Internet connected devices. Online voting systems have a set of security requirements focused on ensuring at least the same properties of traditional voting scenarios. Specifically, in Scytl's systems we provide end to end security, which guarantees that a vote is protected from the very beginning when it is generated in the voter's device until the end of the election when it is decrypted. This requires a specific software in the voters' devices, referred to as the *voting client*, in charge of performing most of the cryptographic operations required to protect the ballot. Our first voting clients were developed as Java Applets. However, in 2013 Scytl decided it was imperative to develop a voting client purely based on Javascript, due to the better multi-platform user experience that this web technology offers and due to the increasing loss of Java support in the browsers.
This industrial paper describes the initial design challenges of the Javascript voting client, the implementation experience and the lessons learned during its development and deployment for our remote electronic voting systems. The paper is complemented with 1) an analysis of the implemented Pseudo-Random Number Generator, 2) a performance study of the main cryptographic primitives used in our voting clients and 3) a performance study of the voting casting process for a given election setup.

**Keywords:** remote electronic voting, JavaScript security, implementation, performance, random number generation

## 1 Introduction

Remote electronic voting systems enable voters to remotely cast votes from their own devices, such as PCs, laptops, smartphones, etc. This is specially suitable for voters who live abroad, who are outside of their region during the election day, and for impaired voters who may have mobility issues.

In general, remote electronic voting systems have to fulfil a set of security requirements in order to be used in electoral processes, which are focused on ensuring that at least the same properties of traditional voting scenarios are maintained, such as vote

authenticity and privacy, result accuracy, secrecy of intermediate results, verifiability and auditability, and uncoercibility and vote selling protection. In order to fulfil such security requirements, remote electronic voting systems use advanced cryptographic protocols (see for example [1], [12], [14]) that usually offer end to end protection of the votes. Due to this, the protocols require to perform several cryptographic operations both at the client and server sides. The piece of software which is run in the voter's device, which is in charge of performing the ballot presentation, navigation and most of the cryptographic operations, is referred to as the *voting client*.

The first voting clients implemented in Scytl's products were developed as Java Applets. The usage of Java enabled 1) the possibility to perform complex cryptographic operations on a multiplatform setup and 2) code and expertise reuse of the developers that were already working on the backend. In addition, Java Applets were at that time the only multi-platform technology that enabled the performance of complex cryptographic operations on the browser. However, the use of Java Applets implied a high price to pay in terms of user experience and security. Voters' devices required a Java Runtime Environment (JRE), that was not always present neither updated and that had become a source of security vulnerabilities. In addition, JRE was not supported in most of the mobile devices. Latterly, its support has recently removed from the most popular browsers. Thus, in the recent years, the natural evolution of the World Wide Web standards and the introduction of HTML5 have strengthened the use of Javascript for increasingly complex operations, reaching a point where performing cryptographic operations in Javascript at the browser has become feasible. Due to this feasibility and the advantages the Javascript technology offered in terms of user experience and multiplatform compatibility (specially for mobile devices), in 2013 Scytl decided it was imperative to develop a voting client purely based on this technology to replace the former Java-based versions.

This industrial paper, which is an extension of a former paper [5] presented at SECRYPT 2016, shows the implementation experiences and lessons learned after the development and deployment of several Javascript voting clients that have been used in several real elections. The paper is organised in seven sections: Section 2 describes the components of a remote electronic voting system, the main components required by a generic voting client, the interaction of this component with the main system and the cryptographic operations it may require; Sections 3 and 4 describe the challenges to implement a Javascript-based voting client and the solutions adopted; Section 5 explains the testing of a pseudo-random number generator implemented; Section 6 does a thorough analysis of the performance of the cryptographic primitives and shows the voting times of the client implemented in Javascript; finally Section 7 presents the conclusions and further work.

## 2 Voting client

This section introduces the details of generic voting clients used in remote electronic voting systems.

## 2.1 Remote electronic voting systems

From a high level point of view, a remote electronic voting system can be divided in two main parts, the *voting servers* and the *voting client* (see Figure 1). During the election, the voting servers provide the back-end services that allow voter authentication, ballot provision, and verification and storage of cast ballots in the ballot box. During the counting phase, the voting servers decrypt and tally the votes providing the election results, while protecting the voter's privacy. A usual approach for this is to perform the processes of cleansing and mixing. The cleansing validates the votes received and removes the voter identity from them, and the mix-net [3] shuffles and transforms the encrypted votes prior to decryption. These two processes break the relation between voters and votes.

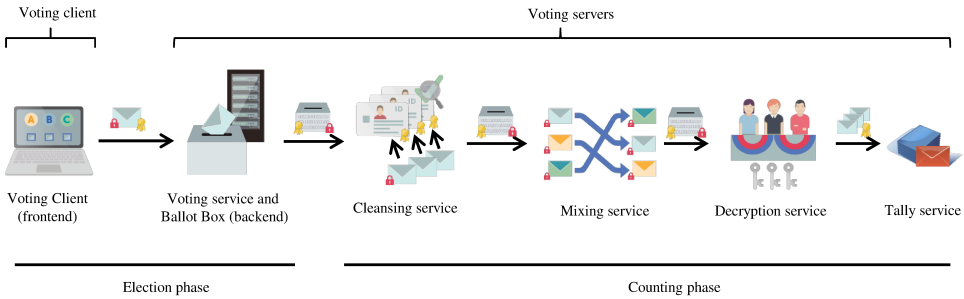


**Fig. 1.** Remote electronic voting system

A good practice in remote electronic voting is to cryptographically protect the vote by encrypting and digitally signing it at the voter's device, before it is sent to the remote voting server for being stored. Hence the privacy and the integrity of the vote are protected from the very beginning, right after the vote is generated until it is processed at the counting stage. This is commonly known as end-to-end encryption and it is possible thanks to the voting client application executed at the voter's device.

The voting client is the front-end that allows the voters to authenticate, navigate through the ballot, select their voting options, generate an encrypted and signed ballot and cast it.

## 2.2 Basic functionalities

The voting client comprises a graphical user interface and some underlying logics. The following functionalities are implemented by the voting client:

– **Authentication**: The voting client deals with the authentication of the voter in front of the voting system. The authentication subsystem can also be adapted for integration with the customer infrastructure or totally replaced by a third-party authentication system.

- **Key management**: The voting scenario requires the voters to have their own cryptographic signing keys to sign their votes, e.g. within a smartcard or installed in their computers. However, in many cases this is unfeasible or unpractical. As a consequence the voting system can include a functionality called *key roaming*, which provides the voter with, at least, a pair of cryptographic signing keys. These keys are delivered by the voting server and protected within a password-sealed container such as PKCS#12 [31]. When key roaming is used, the password required to open the keys is usually derived from the voter credentials. In this case, the credentials are never directly sent to the voting system for authentication. Instead another derivation is used for authentication. This prevents the election servers to have access to the voter keys.
- **Vote generation**: The voting client also generates the ballot to be cast, i.e. it encodes the voting options selected by the voter, encrypts and digitally signs the ballot containing them, and sends the ballot to the remote voting server. Additionally, it may generate mathematical proofs to prove the correctness of the operations performed.
- **Cryptographic library**: The authentication and ballot generation functionalities require the usage of cryptographic primitives that are not natively provided by the Javascript language. Thus they are included as cryptographic libraries.
- **Pseudo random number generator and entropy collector**: A module to generate sequences of secure random numbers, required by some cryptographic primitives, is included for the platforms that do not possess a built-in generator.

### 2.3 Basic voting flow

The flow in the voting client (see Figure 2) depends on the specific voting protocol implemented each one providing different security properties under different assumptions [11], [28]. Despite this, a generic set of steps is common to most of the voting protocols we implement: 1) The voter introduces her credentials to the voting client in order to authenticate; 2) the voting client derives the appropriate identification (Voter ID) and sends it to the server; 3) the server checks the voter is eligible, gathers the voter signing keys and creates an authentication token; 4) then the voting client opens the voter keys and presents the ballot to the voter; 5) then she selects the voting options that represent her voting intent; 6) after the voter confirms her vote, the voting options are encoded and encrypted; 7) mathematical proofs of correct encryption are generated (if required by the protocol); 8) the encrypted voting options and the mathematical proofs are digitally signed; 9) the vote is cast to the remote server; 10) the server computes a vote receipt and its signature; and 11) the voting client validates the receipt signature and presents it to the voter. The receipt allows the voter to check her vote has been received and decrypted by the system, since a list of vote receipts is published at the end of the election. The signature of it proofs the authenticity of the receipt, thus it is not possible to create fake random receipts and pretend the system has lost votes.

### 2.4 Cryptographic functionalities

The following generic set of cryptographic functionalities and algorithms, depending on the voting protocol implemented, may be required in the voting client:

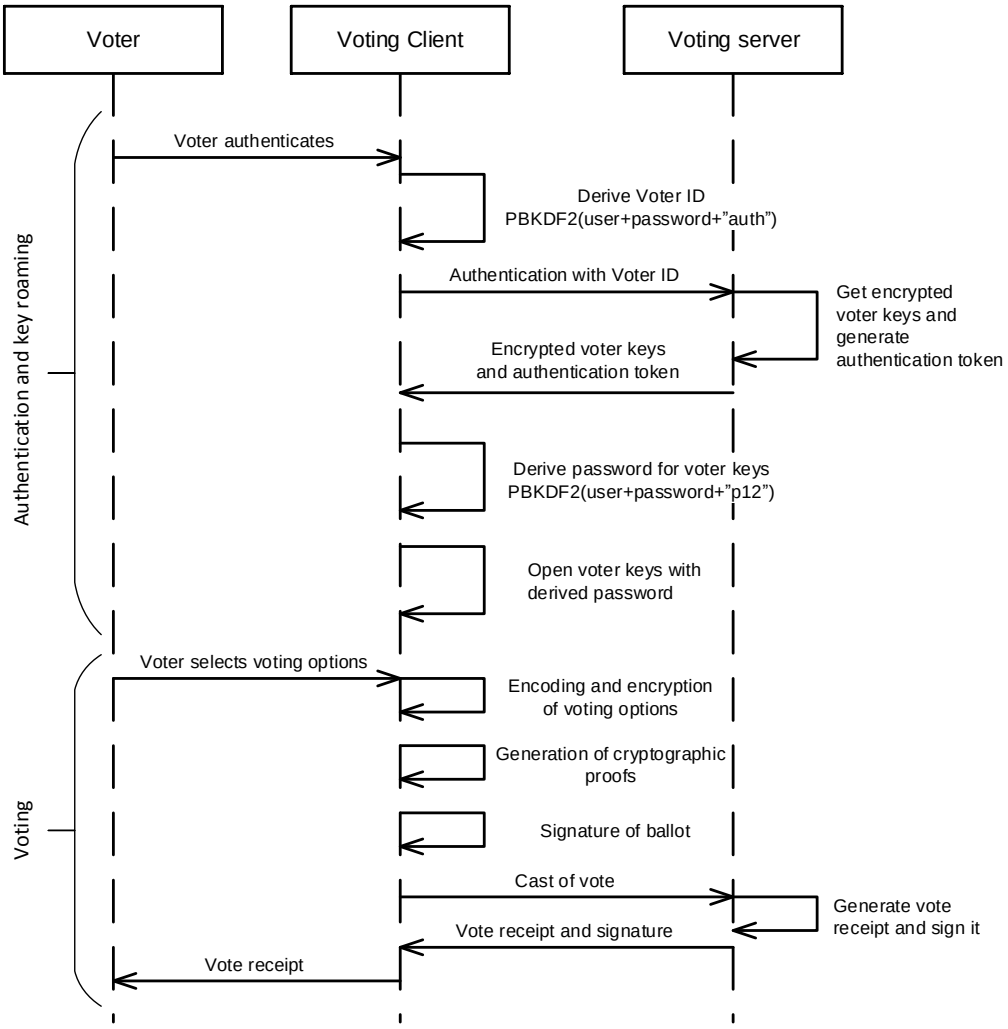


**Fig. 2.** Authentication and voting flowcharts

- **Hash functions**: they are used for computing certificate fingerprinting and digital signatures. Although the standard for SHA-3 has already been published [26], the previous standard SHA-2 [25] (specifically, SHA-256) has been considered due to compatibility issues.
- **Digital signature functions**: they are used for digitally signing the vote and for verifying the digital signatures of the information received from the remote voting server. The RSA-PSS algorithm [29] has been considered.
- **Encryption algorithms**: they are used for encrypting the voting options. RSA-OAEP [29], RSA-KEM and/or ElGamal encryption algorithms [21], and the AES-GCM encryption algorithm [22, 23], are considered for public key and symmetric key cryptography respectively.
- **Zero-Knowledge Proofs of Knowledge (ZKPK)**: they [6] are used to prove a certain statement without revealing any other information than the statement is true. For example the Schnorr Signature [34] can prove knowledge of the randomness used for encrypting a message, providing assurance of the originator of an encrypted vote, and preventing vote copying [4].
- **Pseudo-random number generators (PRNG)**: these are used for generating the random values required by the cryptographic algorithms.
- **Password-based key derivation functions**: in the voting client they are used to derive keys for the authentication of the voter and to access private data. The PBKDF2 [32] algorithm is the one selected.
- **Functionalities for parsing and opening keystores**: these are used for providing private signing and encrypting keys to the voters. PKCS#12 [31] or equivalent key containers are used.
- **Reading, parsing and validation of digital certificates**: these are used to check the validity of the cryptographic keys and X509v3 [30] certificates used in the application.

## 3 Challenges

The implementation of a voting client in Javascript implied several challenges on cryptography, security and performance.

### 3.1 Cryptography

**Availability of cryptographic primitives** A voting client requires to perform cryptographic operations. However, the JavaScript specifications do not include any cryptographic primitives. Instead, some cryptographic functionalities are provided by third party Javascript cryptographic libraries.

Before starting the development of the voting client we performed an analysis of the existing libraries in order to decide which library was more appropriate. The following factors were considered: cryptographic functionalities provided, in order to know what was provided by the library; updates, to know the maintenance level of the code; license, to determine if the license terms were compatible with the ones of the voting client application. In our particular context we preferred BSD, MIT or LGPL licensed libraries.

| **Name** | SJCL | Forge | jsbn | jsrsasign | CryptoJS | asmCrypto |
|---|---|---|---|---|---|---|
| Version | 1.0.6 | 0.6.45 | 1.4 | 6.2.0 | 3.1.8 | 0.0.10 |
| Hashing | SHA-1, SHA-256, SHA-512 | SHA-1, SHA-256, SHA-384, SHA-512 | X | SHA-1, SHA-224, SHA-256, SHA-384, SHA-512 | SHA-1, SHA-224, SHA-256, SHA-512, SHA-3 | SHA-1, SHA-256, SHA-512 |
| RSA signature with SHA-256 | X | RSASSA-PKCS1-V1_5, RSASSA-PSS | X | RSASSA-PKCS1-V1_5, RSASSA-PSS | X | RSASSA-PSS |
| RSA encryption | X | RSAES-PKCS1-v1.5, RSAES-OAEP, RSA-KEM | RSAES-PKCS1-v1.5 | X | X | RSAES-OAEP |
| ElGamal encryption (over Zp) | X | X | X | X | X | X |
| AES encryption | √ | √ | X | X | √ | √ |
| AES cipher modes | GCM, CBC, OCB2, OCB2 progressive, CTR | GCM, CBC, OFB, CFG, CTR | X | X | CFG, OFB, ECB, CTR, CTR Gladman | EBC, CBC, CFB, OFB, CTR, CCM, GCM |
| Schnorr Signature or ZKPKs | X | X | X | X | X | X |
| BigInteger support | √ | √ | √ | √ | X | √ |
| PRNG | √ | √ | √ | X | X | √ |
| Parsing PKCS#12 containers | X | √ | X | X | X | X |
| Parsing X.509 certificates | X | √ | X | √ | X | X |
| PBKDF2 | √ | √ | X | √ | √ | √ |
| **Updates** | $\simeq$ 2 upd/year | +15 upd/year | No updates | +15 upd/year | $\simeq$ 3 upd/year | $\simeq$ 2 upd/year |
| **License** | BSD 2-Clause, GPL-2.0 | BSD 3-Clause, GPL-2.0 | BSD | MIT | MIT | MIT |

**Table 1.** Cryptographic functionalities provided by third party libraries

The results of the analysis, updated on November 2016, are shown on Table 1. This analysis includes an additional library, the asmCrypto, regarding the original one. At the time of developing the solution, in 2013, the conclusion was that the Forge[3] library was one of the best alternatives to implement a voting client. It provided functions for parsing and opening PKCS#12 containers, as well as managing digital certificates and public key cryptography. Besides this, it had a strong developer support and updates were constantly made. Both SJCL[4] [35] and jsbn[5] were also broadly used in other software. Specifically, jsbn was used in other of the analysed libraries to provide BigInteger support (Forge, jrsasign[6]), and in the Helios voting system [1]. SJCL was a library developed by highly recognised cryptographers. Therefore it was worth to consider it for some of the needed functionalities, such as secure random generation or hash algorithms. jsrsasign and CryptoJS[7] did not seem to provide enough support from the point of view of maintenance and use, as well as functionalities. Considering the analysis performed, the Forge library was selected. The functionalities missing in the library were implemented on top of it (see Section 4.1).

The decision to use Forge is still valid considering the updated analysis shown in this article. The only point to consider is the addition of a recent new library, asmCrypto[8], which their authors claim to be faster than the other libraries. However it provides fewer functionalities (some of them required and present in Forge, e.g. RSA-KEM, parsing of PKCS#12 containers and X.509 certificates) and it is updated much less frequently. Another modification to consider, not included in the analysis, is the W3C Candidate Recommendation Web Cryptography API [37] for web browsers. This API includes most of the cryptographic functionalities required by the voting client. However, at the time of writing is still not considered standard, hence it is not guaranteed to be included in all the browsers neither to be included with the same degree of completeness. The advantage of this API is that no external libraries are required and that the computational performance can be better than the Javascript implementations.

**Secure random numbers and entropy** Several cryptographic primitives require the use of random values. The quality of the random values (in the sense of their unpredictability) determine the security offered by these primitives (for example, non-random values can lead to weak encryptions). Although Javascript has a native method for random number generation, *Math.random()* [7], its implementation is not considered cryptographically secure [16]. Alternatively, the W3C Candidate Recommendation Web Cryptography API[37], previously mentioned, includes a Pseudo Random Number Generator (PRNG) suitable for cryptographic uses, *window.crypto.getRandomValues()*. This PRNG uses the browser's host system entropy, but is only implemented in recent versions of the desktop and mobile browsers[9].

[3] http://digitalbazaar.com/forge
[4] http://crypto.stanford.edu/sjcl
[5] http://www-cs-students.stanford.edu/~tjw/jsbn
[6] http://kjur.github.com/jsrsasign
[7] http://code.google.com/p/crypto-js
[8] https://github.com/vibornoff/asmcrypto.js/
[9] https://developer.mozilla.org/en-US/docs/DOM/window.crypto.getRandomValues

Some of the cryptographic libraries analysed have their own PRNG implementations. Specifically, SJCL and Forge implement the Fortuna PRNG by Schneier and Ferguson [8]. This PRNG is intended to be used in long-term systems, such as servers, and provides measures for collecting randomness from the system events, and mixing it in order to provide secure random values. However, web sessions in which Javascript cryptography may be used can be very different from a server system, and therefore the same approaches taken in the design of the Fortuna algorithm may not be the best choice. For example, web sessions have a short life time, and the sources of entropy in a browser or in a server are not the same. Current implementations from SJCL and Forge of Fortuna have modifications regarding the original scheme [35]. However, they still have some limitations regarding the entropy sources they use to generate the random numbers. Thus a custom PRNG, based on the Fortuna PRNG, has been implemented (see Section 4.2).

### 3.2 Security

**Code authenticity** One of the challenges to develop a Javascript voting client is to ensure the authenticity of the code to be executed in the voter's device. A modification of this code by an attacker could have severe implications in the security of the solution, for example compromising the integrity and secrecy of the votes.

A classical solution to ensure the authenticity of the code consists of signing it using public key cryptography and enforcing the validation of this signature in the browser. However, as opposed to other technologies such as Java Applets, there is no standard to sign and verify the Javascript code delivered to the browsers. There only exists a proprietary, and deprecated, solution[10], implemented in Mozilla and, formerly, Netscape Communicator 4.x, browsers. This solution was based on packaging the Javascript and HTML code within a signed Jar file, that was verified by the browser when accessed. More promising is the new Candidate Recommendation *W3C Subresource Integrity* [36] that enables the HTML code to include fingerprints of the Javascript code it refers. Despite this ensures the integrity of the included third party code, it does not guarantee the integrity of the HTML files that contain the hashes. Thus, it is not a complete solution for the authenticity of the whole code of the voting client. Finally, there exist proposals ensuring end to end integrity, but they require the installation of browser plugins [15].

Given the current lack of support for guaranteeing the JS code authenticity, the following approaches, to detect code manipulation, have been implemented: 1) TLS communications to guarantee the code transport authenticity (preventing man-in-the-middle attacks); 2) Regular checks of file integrity in the server, including the Javascript voting client code against a baseline, e.g. using software scanning tools such as AIDE[11]; 3) Running a Javascript remote integrity validation service, i.e an externally executed service to remotely download a selected set of Javascript code from the server as a regular user and check if it matches a previously generated baseline. This service has

[10] http://www.mozilla.org/projects/security/components/ signed-scripts.html
[11] http://aide.sourceforge.net

been implemented and used within the context of an election, but still not to check a voting client code.

**Third party code** Javascript allows the inclusion of third party code and, more important, the dynamic loading of scripts from different servers. However, embedding third party code dynamically loaded from external servers inside the voting application's website can pose a serious security risk and it is totally unadvised, as this code could access any variable or method of the voting application. A server that does not belong to the election realm may not fulfil the same security policies, potentially becoming a weak point of attack. Any legitimate external server acting as code source must be secured as the main code server is.

An example of this is the vulnerability [13] that a team of researchers discovered in the Javascript voting client that we implemented for the State General Elections 2015 of New South Wales[12]. In this case a third party code owned by Piwik, used by monitoring purposes, was included on behalf of NSW. A manipulation of this code, exploiting the FREAK vulnerability present in the external Piwik server that hosted the code, could potentially allow a sophisticated attacker to alter the voting client code running on the voter's browser and modify the intended voting options. From the point of view of the secrecy and integrity of the vote, the reported vulnerability's potential damage was similar to that of having malware installed in the voter's device. This possibility was already considered in the design of iVote 2015, and the defence against it (regarding the integrity of the vote) was the inclusion of a voice verification mechanism using the DTMF phone input.

Thus, we do not recommend including third party code from external servers. But, this may change if the *W3C Subresource Integrity* [36] candidate recommendation becomes an adopted standard.

### 3.3 Performance

The computational performance of Javascript is constantly improving, but still below the one obtained by native applications. In addition, the Javascript applications can be executed in a myriad of devices with very different computational capabilities, e.g. smartphones, laptops, etc. Some existing voting client implementations [1] combined Javascript with the usage of Java for certain primitives that required higher performance using a technology called LiveConnect. However, this alternative could not be considered because the aim was to completely eliminate the dependency with the Java Virtual Machine.

As cryptographic operations are computationally expensive, an efficient implementation was required and several optimisations had to be performed at the cryptographic protocol level (see Sections 4.1 and 4.3) to provide reasonable voting times.

[12] http://www.vote.nsw.gov.au

## 4 Implementation experience

During 2013-2015, most of the Scytl voting systems were transitioned to use a Javascript voting client. This section describes the most relevant aspects of the implementation of them considering the challenges previously described.

### 4.1 Cryptographic library

Scytl provides different voting systems with different cryptographic protocols, thus several variants of the Javascript voting client were implemented. As a consequence, a cryptographic library containing a large amount of primitives was developed:

**Basic primitives** The basic cryptographic primitives are the most widely used in standard web applications and, therefore, present in some of the libraries studied. For example, SHA-256 hash functions, RSA-PSS digital signature, RSA-KEM encryption, AES-GCM symmetric encryption, key derivation functions such as PBKDF2, and support for X.509 certificates and PKCS#12 containers. These primitives have been wrapped in our library from the Forge library.

**ElGamal and ZKPK primitives** Other primitives such as ElGamal encryption or ZKPKs are more specific to cryptographic protocols such as those for e-voting. Therefore, they are not included in the existing libraries. A custom implementation of these primitives has been built.

ElGamal encryption scheme has homomorphic properties that are essential in electronic voting [1], [12], [9]. However, the usage of modular exponentiations with large integers is computationally expensive. Therefore, two modifications were performed to the original scheme to improve its efficiency:

- **ElGamal encryption with short exponents**: This is a well-known optimisation [10, 27] consisting of using shorter exponents (e.g. of about 256 bits) than those defined by the cyclic group used in the scheme (which may be of about 2048 bits). This reduces the cost of the modular exponentiations without posing at risk the security of the scheme in practice [17].
- **ElGamal encryption with multiple keys**: When the number of plaintexts to be encrypted is higher than one, an optimisation consists of reducing the number of exponentiations to compute by using a different public key for computing each ciphertext, but the same randomness for all them. This does not affect the security of the encryption scheme [12, 18], and we show it with an example.

  First, we have to introduce the ElGamal encryption scheme: let the encryption scheme be defined over a subgroup $\mathbb{G}$ of prime order $q$, which has a generator $g$ of elements in $\mathbb{Z}_p^*$. Then, the keypair $(h, x)$ is generated as $x \xleftarrow{\$} \mathbb{Z}_q$, $h = g^x$. A message $m$ is encrypted as $(g^r, h^r \cdot m)$, where $r \xleftarrow{\$} \mathbb{Z}_q$, and decrypted as $m = (h^r \cdot m)/(g^r)$.

Usually, when more than one message has to be encrypted, the direct approach is to use fresh (different) randomness for each new ciphertext to generate. This means that, when encrypting two different messages $m_1$ and $m_2$, the result is: $(g^{r_1}, h^{r_1} \cdot m_1)$, $(g^{r_2}, h^{r_2} \cdot m_2)$. Obviously, given these two ciphertexts an adversary cannot obtain any information about the relation between $m_1$ and $m_2$. For example, if it divides one ciphertext by each other, the relation it can observe is $(g^{(r_1/r_2)}, g^{x(r_1/r_2)} \cdot \frac{m_1}{m_2})$, and by the discrete logarithm problem it cannot obtain $\frac{m_1}{m_2}$, given that $x, r_1, r_2$ are unknown.

On the other side, in the approach we follow by using multiple keys, the result of encrypting $m_1$ and $m_2$ is $(g^r, h_1^r \cdot m_1)$, $(g^r, h_2^r \cdot m_2)$, using the keys $h_1 = g^{x_1}$ and $h_2 = g^{x_2}$ respectively. Still, the information that is leaked to an adversary is the same as in the case before: by dividing the ciphertexts, the adversary observes $g^{r(x_1/x_2)} \cdot \frac{m_1}{m_2}$ which, also given that $x_1, x_2, r$ are secret, cannot be used to obtain information about the relation between $m_1$ and $m_2$.

In order to maximise the code reuse and reduce the likelihood of errors, we used the Maurer framework [20], which generalises the implementation of ZKPKs for different statements. Thus a unique base code is used for all the ZKPK variants of our voting systems. The Java-like BigInteger functionalities required to operate with the large magnitude integers used in these primitives were reused from the existing libraries.

### 4.2 Pseudo-random number generator

A pseudo-random number generator (PRNG) is an algorithm that generates a sequence of numbers which is cryptographically indistinguishable from a sequence of true random numbers. PRNGs generate the sequence of numbers in a deterministic manner. The unpredictability of the values generated by a PRNG is given by the unpredictability of the value with which it is initialised, which is called the seed. In order to provide high quality random values for the cryptographic primitives, the PRNG is seeded with entropy (random data) from the system events and information.

We have implemented a custom PRNG to be used in the voting client. Its design has taken into account requirements and particularities of voting clients, specifically a short runtime life and strong unpredictability of the random values produced. The following principles were followed in the design and use: 1) the collection of entropy to seed the PRNG had to start as soon as possible, preferably as soon as the voter started interacting with the system. The implemented PRNG provides methods to start the entropy collection task before the protocol-specific functions have to be called; 2) in order to collect entropy as fast as possible, user-driven events from an extense set of sources were collected; 3) entropy estimation mechanisms were included in order to estimate how much entropy can be attributed to each type of information collected in the browser. These mechanisms ensure that enough entropy is collected for seeding the PRNG and starting generating secure random values.

The PRNG functionality implemented is composed by several parts (see Figure 3) detailed in the following sections.

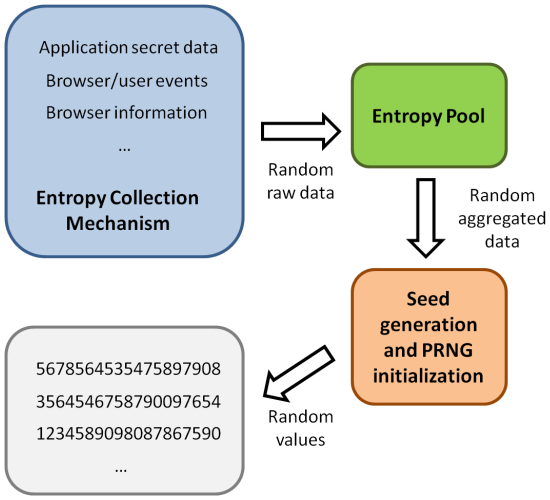


**Fig. 3.** Diagram of the Pseudo-Random Number Generator

*Entropy collection mechanism* Collects information available in the browser, as well as events generated by web navigation and user interaction:

- Sources of information available in the browser or in the voting application.
  - Random numbers from Javascript cryptographic API if available in the used browser
  - Browser information (e.g. User Agent string)
  - Date
  - Regular random numbers from the standard Math library
- Events triggered by JavaScript or Ajax calls, as well as user events.
  - Mouse: *mousemove*, *mousedown*, *mouseup*, *wheel*
  - Keyboard: *keydown*, *keyup*
  - Touch: *touchstart*, *touchmove*, *touchend*, *gesturestart*, *gesturechange*, *gestureend*
  - Device accelerometer/compass: *devicemotion*, *deviceorientation*
  - Page loads and other AJAX calls

Since the Javascript cryptographic API is a source of cryptographically strong random values, this collector is defined to be the first one to be called. In user events the following information is collected when possible: relative position to the window and to the screen, key/button, date, angle (only for compass) and acceleration components (x,y,z) (only for accelerometer in smartphones, tablets and alike). Each event collected has been assigned with an estimated entropy value, thus it is possible to estimate the amount of entropy gathered by the entropy collector. Most of the estimated entropy values were obtained from existing analysis [35].

*Seed generation and initialisation* The data gathered by the entropy collection mechanism is accumulated into an entropy pool, and at the same time an entropy counter is increased with the estimated entropy. The entropy collection starts at the very beginning after downloading the voting client. Every time new data is collected, there is the possibility to check if a minimum amount of entropy required by the application has been reached. After collecting enough entropy the collector mechanism is stopped and the PRNG is seeded. An amount of 128 bits of entropy is enough for initialising the PRNG.

*Pseudo-random values generation* The PRNG generates sequences of pseudo-random values suitable for cryptographic purposes. Our implementation is based on the Fortuna PRNG [8] and the random values are obtained from an AES cipher in counter mode. The PRNG output has been tested against statistical analysis tools (see Section 5) to guarantee its robustness.

### 4.3 Vote generator

The Javascript implementation of the voting client has influenced the voting protocols because the computational efficiency is limited and the trust of the underlying platform cannot be guaranteed. An overview of the main considerations is shown below.

*Pre-computations* The idea behind this is to pre-compute some cryptographic operations' values while the voter selects the voting options, so that when the voter decides to cast the vote the number of remaining computations is small and the time the voter has to wait for the ballot to be cast is considerably shorter. The amount of operations that can be pre-computed can account up to 95% in certain voting protocols.

An example of pre-computation of ElGamal encryption is the following: considering the ElGamal encryption scheme defined in 4.1, the most costly operations required to encrypt a message (which are two exponentiations), can be done before the message is available: $(g^r, h^r)$. After that, the remaining operation is a product in order to obtain the encryption of $m$, $(g^r, h^r \cdot m)$. Since the product is a much more cheaper operation than the exponentiation (moreover given the length of the exponents), there is a significant gain in computation time. Similar adaptations can be done to the ZKPKs, where the exponentiations can be pre-computed and the remaining operations to be done once the values to prove are available consist mainly on hashes (usually fast) and products.

*Cast as intended validation* Since the voter device cannot be trusted, e.g. it may be infected by malware, some of the voting protocols implemented provide validation mechanisms that allow the voter to check the integrity of the vote cast. For example, in the Norwegian project eValg2013 [12] secret codes received via SMS allowed voters to check that the votes cast contained the voting options they selected. In the Neuchâtel e-voting platform [9], the same approach was followed with the difference that the codes were returned by the same voting client. This option was possible in this case because multiple voting was not allowed, thus the voting client could not learn the codes and vote again with manipulated voting options.

*Defence against user manipulation* Development tools provided by browsers make the manipulation of the code easier for a regular user than in other languages. A non honest voter could take advantage of this by manipulating the processes that happen on her browser to disrupt the election. As a general rule, it is recommended to make at server-side some pre-processing over the data received from the voting client prior to passing it to further layers of the protocol. In addition, voting protocols must be designed to cope with these cases, e.g. including ZKPKs to be verified at the voting server when the vote is received.

## 5 Test of the PRNG

Since the PRNG is required to produce values with strong randomness, we tested its output against a statistical analysis tool. The tests proved that the implemented PRNG provides random values of the expected quality.

### 5.1 Testing tool

Dieharder [2] was the tool selected to perform the statistical analysis, a random number generator testing suite, intended to test generators. It includes tests from the original Diehard Battery of Tests of Randomness [19], as well as tests from the Statistical Test Suite (STS) [33] developed by the National Institute for Standards and Technology (NIST) and tests developed by the author of the Dieharder test suite. Dieharder is well known and highly reputed due to the broad characteristics of random number generators that are evaluated in its tests.

### 5.2 Test-bed setup

In order to test the PRNG, most of the tests available in the Dieharder suite have been performed, except those which need an overwhelming quantity of data (more than 4GB). Most of the discarded tests overlapped with other tests, thus it was considered not to affect the quality of the testing. Tests which supported different configurations were performed using different input parameters, thus in total an amount of 78 statistical tests were performed.

In order to have a reference of what should be expected from the tests on the implemented Scytl PRNG, two other PRNGs have also been evaluated in the same way: a Flawed PRNG which is known to generate sequences of correlated random numbers, and the AES_OFB generator as the Gold Standard PRNG, which is a commonly known good PRNG.

A set of datasets were generated for each of the PRNGs to test (11 datasets for the Scytl PRNG, 5 datasets for the Gold Standard PRNG and 3 datasets for the Flawed PRNG). Each dataset contained 256 million random unsigned 32-bit integers, which was composed of 400 sets of 640.000 values that were generated with a different PRNG instance and seed. These values composed the input of the Dieharder test suite. Values generated with the PRNG initialised with different seeds have been used in order to

test, not only that the values sequentially generated by a PRNG instance have the expected properties (corresponding to a sequence of random numbers), but also to test that different instances of the PRNG (initialised with different seeds) generate uncorrelated random numbers.

### 5.3 Results

All the tests have been successfully passed by the implemented Scytl PRNG, as well as by the Gold Standard PRNG (see Table 2). On the other hand, the Flawed PRNG has failed most of the tests.

| **Dieharder tests performed** | **Gold PRNG** | **Scytl PRNG** | **Flawed PRNG** |
|---|---|---|---|
| Diehard Birthdays Test | ✓ | ✓ | ✓ |
| Diehard 32x32 Binary Rank Test | ✓ | ✓ | X |
| Diehard 6x8 Binary Rank Test | ✓ | ✓ | X |
| Diehard Bitstream Test | ✓ | ✓ | ✓ |
| Diehard OPSO | ✓ | ✓ | X |
| Diehard OQSO Test | ✓ | ✓ | X |
| Diehard DNA Test | ✓ | ✓ | X |
| Diehard Count the 1s (stream) Test | ✓ | ✓ | ✓ |
| Diehard Count the 1s Test (byte) | ✓ | ✓ | X |
| Diehard Parking Lot Test | ✓ | ✓ | ✓ |
| Diehard Min. Dist. (2d Circle) Test | ✓ | ✓ | ✓ |
| Diehard 3d Sphere (Min. Dist.) Test | ✓ | ✓ | ✓ |
| Diehard Squeeze Test | ✓ | ✓ | X |
| Diehard Sums Test | ✓ | ✓ | ✓ |
| Diehard Runs Test | ✓ | ✓ | ✓ |
| Diehard Craps Test | ✓ | ✓ | X |
| STS Monobit Test | ✓ | ✓ | X |
| STS Runs Test | ✓ | ✓ | X |
| STS Serial Test (Generalized) | ✓ | ✓ | X |
| RGB Bit Distribution Test | ✓ | ✓ | X |
| RGB Generalized Min. Dist. Test | ✓ | ✓ | X |
| RGB Permutations Test | ✓ | ✓ | X |
| RGB Lagged Sum Test | ✓ | ✓ | X |
| RGB Kolmogorov-Smirnov Test Test | ✓ | ✓ | X |

**Table 2.** Dieharder tests performed

In order to determine the quality of a PRNG, Dieharder tests the *null hypothesis*, which is that the sequence of numbers generated by the random number generator under test is *truly random*. Dieharder computes certain statistics over the random values generated by the PRNG, which ultimately lead to the p-value. This value denotes the probability that a true random number generator would produce a sequence of similar characteristics. That is, it summarises the evidence against the null hypothesis: if the p-value is lower than a certain significance level, the null hypothesis is rejected and the PRNG under test is not accepted as a good one. For other p-values, the null hypothesis is not accepted, but it fails to be rejected for this particular test. In fact, for good PRNGs, p-values extracted from different rounds of each test are expected to be uniformly dis-

tributed. More information about the null hypothesis and the p-values can be found in [24] and the Dieharder manuals[13].

More explicitly, in order to determine whether a test succeeds or fails, Dieharder internally calculates a set of p-values performing several executions of each test with different data. If the p-values of a given test are smaller than the significance level (usually values in the range of 0.1 to 0.001), then the test fails. In addition, the set of p-values obtained are tested with the Kolmogorov-Smirnov (KS) test to check the null hypothesis is not rejected. The result of this check is a number between 0 and 1.

Since several datasets were generated for each PRNG, each Dieharder test was repeated a few times for each PRNG. In order to test the different runs of each test, we have treated the KS test values output for each test as p-values. Then, we have set a significance level of 0.005 and we have considered the test failed if any of the values were below this threshold. Another issue to be considered is that these values are random variables and, as such, the same test over different datasets should produce noticeably different values. This is why we compared the difference between the maximum and minimum values, and considered the test failed if this difference was smaller than 0.1. Both Scytl 's and the Gold Standard PRNG succeed with this test as well, in particular such difference was much bigger than 0.1 for all the tests. On the other hand, the flawed PRNG had many similar values, and as a consequence it failed most of the tests.

## 6 Performance

A good performance of the voting client application is important to guarantee a smooth user experience. The next sections study the timing associated to the voting client and its cryptographic operations.

### 6.1 Cryptographic operations

A benchmarking application was developed to measure the speed of some cryptographic operations provided by our Javascript cryptographic library. Most of the operations tested are used by our voting clients. Others are not used, but they are included as a reference (for example the ElGamal decryption primitives). The benchmarks were performed on a PC with several operating systems and browsers and on two smartphones based on Android and iOS operating systems. The results obtained (see Table 3) are product of one execution of the benchmark application on each of the systems detailed. The time shown for each operation is the average of 1000 executions for the SHA-256, HMACwithSHA256 and AES-128-GCM operations, 100 executions for the RSA ciphers and signers, and 10 executions for the PBKDF2, ElGamal cipher and SchnorrProof generator and validators. The PBKDF2 is setup with 32.000 iterations and computes a key of length 128 bits. All the symmetric and asymmetric keys used in the tests are 128 and 2048 bits long, respectively. The ElGamal primitives are tested both with and without the optimisations described in Section 4.1. The Schnorr primitive is only

[13] http://manpages.ubuntu.com/manpages/precise/man1/dieharder.1.html

| **Device** | **PC i5-3210M CPU 2,50GHz, 8GB RAM** | | | | | | **SG S6 Edge** | **A iPhone 6** |
|---|---|---|---|---|---|---|---|---|
| **OS** | **Windows 10 Professional** | | | | **Ubuntu 16.04.1 LTS** | | **Android 6.0.1** | **iOS 10.0.2** |
| **Browser** | **Chrome 54** | **Firefox 50** | **IE 11** | **Edge 38** | **Chrome 54** | **Firefox 50** | **Chrome 54** | **Safari 10** |
| **SHA-256 1KB** | 0,116 | 0,158 | 0,349 | 0,322 | 0,132 | 0,139 | 0,388 | 0,323 |
| **SHA-256 10KB** | 0,521 | 1,168 | 3,085 | 2,832 | 0,533 | 0,905 | 1,743 | 2,083 |
| **HMAC with SHA256 1KB** | 0,244 | 0,479 | 0,809 | 0,825 | 0,226 | 0,199 | 0,647 | 0,399 |
| **HMAC with SHA256 10KB** | 1,529 | 4,547 | 7,125 | 6,989 | 1,387 | 1,802 | 3,886 | 3,258 |
| **PBKDF2 with SHA256 20B 32K It** | 238 | 593 | 1606 | 921 | 250 | 520 | 925 | 971 |
| **AES-GCM-128 1KB Enc / Dec** | 0,429 / 0,447 | 0,454 / 0,574 | 1,141 / 1,178 | 0,533 / 0,655 | 0,475 / 0,511 | 0,504 / 0,684 | 1,201 / 1,253 | 0,637 / 0,748 |
| **AES-GCM-128 10KB Enc / Dec** | 2,301 / 2,998 | 2,298 / 3,845 | 6,460 / 7,457 | 2,904 / 3,859 | 2,558 / 3,214 | 2,927 / 4,944 | 6,849 / 8,120 | 3,781 / 5,888 |
| **RSA-OAEP 214B Enc / Dec** | 2 / 43 | 3 / 51 | 2 / 56 | 2 / 48 | 2 / 47 | 3 / 56 | 4 / 84 | 3 / 67 |
| **RSA-KEM 1KB Enc / Dec** | 3 / 44 | 3 / 48 | 6 / 57 | 6 / 47 | 3 / 42 | 3 / 51 | 8 / 83 | 5 / 67 |
| **RSA-KEM 10KB Enc / Dec** | 5 / 46 | 5 / 51 | 14 / 72 | 8 / 57 | 5 / 45 | 5 / 55 | 13 / 90 | 9 / 73 |
| **RSA-PSS 1KB Sign / Verify** | 43 / 1 | 48 / 2 | 71 / 2 | 52 / 3 | 42 / 1 | 50 / 2 | 82 / 3 | 67 / 2 |
| **RSA-PSS 10KB Sign / Verify** | 45 / 2 | 48 / 3 | 65 / 5 | 54 / 5 | 43 / 2 | 52 / 3 | 85 / 5 | 74 / 4 |
| **ElGamal Enc / Dec (Normal Exp)** | 252 / 444 | 304 / 157 | 353 / 177 | 288 / 147 | 250 / 438 | 316 / 165 | 476 / 238 | 448 / 502 |
| **ElGamal Enc / Dec (Short Exp)** | 115 / 59 | 40 / 20 | 46 / 23 | 34 / 24 | 113 / 61 | 42 / 21 | 62 / 32 | 131 / 66 |
| **ElGamal Enc / Dec (Short Exp + Batch 2e)** | 173 / 117 | 59 / 39 | 68 / 48 | 63 / 40 | 170 / 116 | 63 / 43 | 93 / 197 | 193 / 132 |
| **ElGamal Precomp / Enc (Short Exp)** | 115 / 0 | 40 / 0 | 52 / 0 | 36 / 0 | 113 / 0 | 42 / 0 | 108 / 0 | 130 / 0 |
| **Schnorr Proof Gen / Ver (Short Exp)** | 62 / 112 | 22 / 38 | 27 / 52 | 26 / 40 | 61 / 111 | 24 / 45 | 44 / 67 | 70 / 138 |
| **Schnorr Proof Precomp / Gen (Short Exp)** | 58 / 3 | 20 / 1 | 23 / 2 | 20 / 2 | 57 / 3 | 21 / 1 | 31 / 4 | 65 / 4 |

**Table 3.** Cryptographic operations performance (time in ms)

tested with the version that uses short exponents, which is the one used in the company's voting clients.

From the performance results obtained we can conclude the following:

- The performance is heavily influenced by the type of device and underlying hardware, e.g. a regular PC is faster than a smartphone. As expected, the regular PC performs faster since it is more powerful than a mobile device. However, we have observed that in the last years the distance between both types of platforms has shortened (see for example the results we presented in [5]).
- There are large differences of performance, for certain primitives, comparing different browsers, e.g. SHA256, HMAC, PBKDF2 and AES are slower in Microsoft Explorer and Microsoft Edge browsers, while the ElGamal and SchnorrProof primitives run slower in Chrome browsers for PC. An extreme example is the PBKDF2, that can present differences of up to 7 times.
- Instead, the operating system does not have a strong impact on the results, e.g. similar results are obtained in Linux and Windows with the same browsers.
- Regarding the operation types, the times obtained for the different types of primitives is the one expected, i.e. the following operations spent more time from left to right: hashes, MACs, symmetric encryption, asymmetric encryption and signature based on RSA, and ElGamal encryption and related ZKPs.
- The results also prove the optimisations of ElGamal encryption explained in Section 4.1 are successful. The implementation based on short exponents provides a 5-7 times speedup, the batch encryption spends a 25% less time when two elements are encrypted, and the pre-computations make the final encryption time to be almost negligible.

The results obtained are consistent with the expectations. However, it can be observed that in some cases there are particular primitives that show unexpected timings in a given browser (even in the same browser for a different operating system). For example, the ElGamal decryption without optimisations spends more time decrypting than encrypting in Chrome, when these times should be inverted due to the theoretical complexity of the primitives. This can be related to bugs or faulty optimisations of the Javascript interpreters of the browsers. In some cases, these type of issues can also affect the functionality of the primitives. For example, due to a faulty optimisation of Safari, the Forge library returned incorrect values for certain operations after releasing Safari 10 [14]. A workaround had to be implemented by Forge in order to overcome the problem while Apple did not issue a bug fix release of their browser.

### 6.2 Voting client application

The performance of the voting client may present a high variability depending of:

- **Voting protocol**: defines the cryptographic primitives and communication handshakes performed with the server.

[14] `https://github.com/digitalbazaar/forge/issues/428`

– **Election and cryptographic parameters**: The election parameters define the number of candidates, parties and contests of an election, implying different vote lengths and number of encryptions. The cryptographic parameters define the length of the keys and algorithms used, which influences the timings of the cryptographic primitives.
– **Browser**: Certain browser Javascript engines are much more efficient than others executing the cryptographic operations implemented.
– **Device**: Voter device's processor and memory considerably affect the performance.

The tests performed were focused on measuring the user experience in different browsers and devices for a given voting protocol and set of election parameters. The results reflect the time passed since the voter requests the cast of the vote until the process finishes. Pre-computed operations are not considered since they are executed before the mentioned process.

**Neuchâtel e-voting protocol** The selected voting protocol is the one used in a recent election in Neuchâtel e-voting platform [9] (March 2015). A test election was setup where the voters had to vote for two contests. In the first contest the voter had to choose one party and 5 candidates, whereas in the second contest the voter had to choose one party and 41 candidates. The operations computed by this voting protocol are summarised here:

1. **Encryption of the selected voting options:** the voting options selected of both contests are multiplied together and the result is encrypted into one ciphertext. The ciphertext is computed using the ElGamal encryption algorithm, which requires 2 exponentiations which have been pre-computed while the voter navigates through the application (see Section 4.3).
2. **Computation of partial return codes:** these are codes used to provide cast as intended verifiability (see Section 4.3). The computation of partial return codes requires one exponentiation of each voter selection to a voter-specific secret key, which in the test election sums up to 47 exponentiations. Every time the voter selects an option, the corresponding partial return code is computed. Therefore, these operations are already done when the voter pulses the *send* button. Thus this time is not included in the presented test results.
3. **Encryption of partial return codes:** the partial return codes are encrypted using the ElGamal encryption algorithm, under the public key of the server, in order to protect their privacy during their transmission from the voting client to the server. The variant with multiple key encryption, as described in Section 4.1, is used. The required exponentiations, which sum up to 47, are also pre-computed while the voter navigates through the application.
4. **Generation of cryptographic proofs (ZKPKs):** two ZKPKs are computed. The first, based on the Schnorr [34] identification protocol, proves the encrypted ballot is well-formed. The second, based on the Chaum-Pedersen [3] protocol, prove to the server that the partial return codes match the voting options of the encrypted ballot. For the first proof 1 exponentiation, which can be pre-computed, is required. For the second proof, 7 exponentiations are required, of which 5 can be pre-computed.

Thus, in total 8 exponentiations are computed during the proof generation process, from which 6 can be pre-computed.

5. **Signature of the computed values:** all the computed values are digitally signed using the RSA digital signature algorithm.

The ElGamal encryption and the RSA signature algorithms use 2048 bit keys. The approach described in Section 4.1 for using short exponents (256-bit exponents instead of 2047-bit ones) in ElGamal is used for the encryption of the voting options, for the encryption of partial return codes, and for the Schnorr and plaintext equality proofs.

| Device | OS | Browser | Time |
|---|---|---|---|
| PC i5-3210M CPU 2,50GHz 8GB RAM | Windows 10 Professional | Chrome 54 | 12s |
| | | Firefox 50 | 9s |
| | | IE 11 | 9,5s |
| | | Edge 38 | 8s |
| | Ubuntu 16.04.1 LTS | Chrome 54 | 12s |
| | | Firefox 50 | 9,5s |
| Samsung Galaxy 6 Edge | Android 6.0.1 | Chrome 54 | 18s |
| Apple iPhone 6 | iOS 10.0.2 | Safari 10 | 43,5s |

**Table 4.** Vote casting and confirmation performance

**Results obtained** The results (see Table 4) are aligned with the ones obtained for the cryptographic primitives, although in this case they also include the server and network processing times. The platform is the element that influence the most the results, clearly showing that desktop computers perform much faster. The browser also influences them, being Google Chrome the slowest browser casting a vote in the PC category. But what it is more relevant is that in most of the tested devices the time needed for casting a vote is always below 45 seconds, and usually much less, clearly demonstrating the Javascript technology is appropriate for implementing a voting client. Notice that, in comparison with the previous paper [5], this results have been updated and they show higher times. The reason is that they currently include the time spent by casting the vote and, also, by confirming it (this last part not included in the former paper).

### 6.3 Performance challenges

During the development, one of the first points that arose was that, the execution of cryptographic primitives could be slow on certain browsers and/or platforms. This generated two effects: a) the browser presented a pop-up indicating the script was not responding and b) the time to generate the encrypted ballot was too long. Two solutions were applied to the first issue. The HTML5 Web Workers[15] were used if available. This

[15] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

prevented the script from blocking the view of the page and no pop-up was presented to the user. For browsers no compatible with this technology, the solution implemented consisted of dividing the costly operations in smaller operations, thus avoiding exceeding the timeout associated to the pop-up. For the second issue, the overall time taken to generate a ballot, the primitive optimisations described in Section 4.1, i.e. ElGamal encryption with short exponents and multiple keys, and the pre-computation of part of the cryptographic operations described in Section 4.3 were applied. A second point was related to the generation of random numbers. The amount of entropy we initially required to seed the PRNG was 128 bits. This is recommended, in order to ensure that the random generation is strong enough (i.e. the random values generated are unpredictable). Nevertheless, in order to prevent usability issues due to exceptional cases where the minimum entropy cannot be reached, a minimum amount of entropy collection was not enforced in production. Instead, the entropy estimation mechanism was used only during development time to perform tests that ensured a minimum level of entropy was reached before the first random values were requested.

## 7 Conclusions

In this paper, a revised version of [5], we have explained our experience and lessons learnt on implementing a Javascript voting client in an industrial environment. The resulting voting client has been extensively used in real elections with a very positive outcome.

The implementation described in this article has allowed us to conclude with several outcomes. First, the implementation of a voting client in Javascript is feasible, both from a security and a performance perspective. The language is powerful enough to implement cryptographic primitives and the current interpreters good enough to support the load, i.e. to perform the tasks in the expected amount of time. Second, the usage of Javascript has provided a much better user experience and larger interoperability than previous implementations of the voting client based on Java applets. The reason is that the Javascript voting clients are much lighter and multi-platform than their Java counterparts. No Java Runtime Environment (JRE) for each specific platform is needed, only a browser with Javascript support. This has dramatically reduced the number calls related to the JRE in the customer support service. At the same time, the number of devices where the client can be run is much larger. On the downside, the usage of Javascript has increased the number of code interoperability issues to consider during development related to each type of browser (Firefox, Google Chrome...). However, this does not impact the final user since it is tackled at development time. Also, as there are multiple browser implementations, the possibility of incorrect functioning due to browsers' bugs is higher. Third, a strong security level is offered, thus the cryptographic primitives used are the same than in the Java voting client implementation. In addition, our client does not suffer from the, increasingly, critical security bugs coming up from the JRE. The only disadvantage of Javascript is that there is no support for signing the code. However, the integrity of the code is guaranteed by the transport layer (SSL/TLS) and by file integrity mechanisms at the server. Additional security measures mitigate this issue, e.g. remote code integrity validation services and use of verifiable voting

protocols allowing the voter to verify the vote cast with independence of the voting client logics.

Further work is being performed to deal with the Javascript weaknesses, mostly the lack of code signed support, and performance improvement. On the one hand, an intelligent application for remote code integrity validation service is being implemented. This application issues requests, which should not be distinguishable from regular requests issued by real voters, to retrieve and validate the code served. On the other hand, the cryptographic library implemented is being revised in order to obtain better performance. This is specially important to improve the voting times in resource constrained devices, such as low-end smartphones or smart television platforms. This is endeavoured both by a) studying the performance of new cryptographic libraries targeted to obtain better performance, such asmCrypto; and b) studying the adoption of the Web Cryptography API [37] to be used when supported.

## Acknowledgements

We would like to thank our colleagues David Salvador, Pol Valletbó, Adrià Rodríguez and Anna Mayola for their help with the performance test environments.

## References

1. Adida, B.: Helios: Web-based open-audit voting. In: van Oorschot, P.C. (ed.) USENIX Security Symposium. pp. 335–348. USENIX Association (2008)
2. Brown, R.G., Eddelbuettel, D., Bauer, D.: Dieharder: A random number test suite. Duke University Physics Department (2009)
3. Chaum, D., Pedersen, T.P.: Wallet databases with observers. In: Brickell, E.F. (ed.) Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings. Lecture Notes in Computer Science, vol. 740, pp. 89–105. Springer (1992)
4. Cortier, V., Smyth, B.: Attacking and fixing Helios: An analysis of ballot secrecy. In: Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011. pp. 297–311. IEEE Computer Society (June 2011)
5. Cucurull, J., Guasch, S., Galindo, D.: Transitioning to a javascript voting client for remote online voting. In: Proceedings of the 13th International Joint Conference on e-Business and Telecommunications - Volume 4: SECRYPT,. pp. 121–132 (2016)
6. Damgaard, I.: On $\sigma$-protocols. Cryptologic Protocol Theory, CPT 2010, v.2 (2010), `http://www.cs.au.dk/~ivan/\\Sigma.pdf`
7. ECMAScript: ECMAScript ® Language Specification 5.1 Edition (June 2011)
8. Ferguson, N., Schneier, B.: Practical Cryptography. John Wiley & Sons, Inc., New York, NY, USA, 1 edn. (2003)
9. Galindo, D., Guasch, S., Puiggalí, J.: 2015 Neuchâtel's cast-as-intended verification mechanism. In: Haenni, R., Koenig, R.E., Wikstroḿ, D. (eds.) E-Voting and Identity, Lecture Notes in Computer Science, vol. 9269, pp. 3–18. Springer International Publishing (2015)
10. Gennaro, R.: An improved pseudo-random generator based on the discrete logarithm problem. J. Cryptology 18(2), 91–110 (2005), `http://dx.doi.org/10.1007/s00145-004-0215-y`

11. Gharadaghy, R., Volkamer, M.: Verifiability in electronic voting - explanations for non security experts. In: Krimmer, R., Grimm, R. (eds.) Electronic Voting 2010, EVOTE 2010, 4th International Conference, Co-organized by Council of Europe, Gesellschaft für Informatik and E-Voting.CC, July 21st - 24th, 2010, in Castle Hofen, Bregenz, Austria. LNI, vol. 167, pp. 151–162. GI (2010)
12. Gjosteen, K.: The norwegian internet voting protocol. ePrint (August 2013), eprint.iacr.org/2013/473.pdf
13. Halderman, J.A., Teague, V.: The new south wales ivote system: Security failures and verification flaws in a live online election. In: Haenni, R., Koenig, E.R., Wikström, D. (eds.) E-Voting and Identity: 5th International Conference, VoteID 2015, Bern, Switzerland, September 2-4, 2015 Proceedings. pp. 35–53. Springer International Publishing (2015)
14. Juels, A., Catalano, D., Jakobsson, M.: Coercion-resistant electronic elections. In: Chaum, D., Jakobsson, M., Rivest, R.L., Ryan, P.Y.A., Benaloh, J., Kutylowski, M., Adida, B. (eds.) Towards Trustworthy Elections, New Directions in Electronic Voting. Lecture Notes in Computer Science, vol. 6000, pp. 37–63. Springer (2010)
15. Karapanos, N., Filios, A., Popa, R.A., Capkun, S.: Verena: End-to-end integrity protection for web applications. In: 2016 IEEE Symposium on Security and Privacy (to appear). pp. 895–913 (2016)
16. Klein, A.: Temporary user tracking in major browsers and Cross-domain information leakage and attacks (2008), september-November, 2008, Trusteer
17. Koshiba, T., Kurosawa, K.: Short exponent diffie-hellman problems. In: Bao, F., Deng, R.H., Zhou, J. (eds.) Public Key Cryptography - PKC 2004, 7th Int. Workshop on Theory and Practice in Public Key Cryptography. Lecture Notes in Computer Science, vol. 2947, pp. 173–186. Springer (2004)
18. Kurosawa, K.: Multi-recipient public-key encryption with shortened ciphertext. In: Naccache, D., Paillier, P. (eds.) Public Key Cryptography, 5th Int. Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002. Lecture Notes in Computer Science, vol. 2274, pp. 48–63. Springer (2002), http://dx.doi.org/10.1007/3-540-45664-3_4
19. Marsaglia, G.: Diehard: a battery of tests of randomness (1996), http://stat.fsu.edu/pub/diehard/
20. Maurer, U.: Unifying zero-knowledge proofs of knowledge. In: Preneel, B. (ed.) Progress in Cryptology AFRICACRYPT 2009, Lecture Notes in Computer Science, vol. 5580, pp. 272–286. Springer Berlin Heidelberg (2009)
21. Menezes, A.J., Vanstone, S.A., Oorschot, P.C.V.: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA, 1st edn. (1996)
22. NIST: Federal Information Processing Standard (FIPS) 197, Advanced Encryption Standard (AES). Tech. rep., U.S. Department Of Commerce (Nov 2001)
23. NIST: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, NIST Special Publication 800-38D. Tech. rep., U.S. Department Of Commerce (Nov 2007)
24. NIST: A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications, NIST Special Publication 800-22rev1a. Tech. rep., U.S. Department Of Commerce (Apr 2010)
25. NIST: Federal Information Processing Standard (FIPS 180-4), Secure Hash Standard. Tech. rep., U.S. Department Of Commerce (Mar 2012)
26. NIST: Federal Information Processing Standard (FIPS) 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Tech. rep., U.S. Department Of Commerce (Aug 2015)
27. van Oorschot, P.C., Wiener, M.J.: On diffie-hellman key agreement with short exponents. In: Maurer, U.M. (ed.) Advances in Cryptology - EUROCRYPT '96, Int.l Conf. on the The-

ory and Application of Cryptographic Techniques. Lecture Notes in Computer Science, vol. 1070, pp. 332–343. Springer (1996)
28. Puiggalí, J., Chóliz, J., Guasch, S.: Best practices in internet voting. In: NIST: Workshop on UOCAVA Remote Voting Systems. Washington DC, August 2010.
29. RFC-3447: Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1 (2003)
30. RFC-5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List Profile (2008)
31. RSA Laboratories: PKCS #12: Personal Information Exchange Syntax Standard
32. RSA Laboratories: PKCS #5: Password-Based Cryptography Standard
33. Rukhin, A., Soto, J., Nechvatal, J., Barker, E., Leigh, S., Levenson, M., Banks, D., Heckert, A., Dray, J., Vo, S., Rukhin, A., Soto, J., Smid, M., Leigh, S., Vangel, M., Heckert, A., Dray, J., Iii, L.E.B.: NIST Special Publication 800-22 Rev 1a: Statistical test suite for random and pseudorandom number generators for cryptographic applications (2010)
34. Schnorr, C.P.: Efficient signature generation by smart cards. J. Cryptol. 4(3), 161–174 (Jan 1991)
35. Stark, E., Hamburg, M., Boneh, D.: Symmetric cryptography in JavaScript. In: ACSAC. pp. 373–381. IEEE Computer Society (2009)
36. W3C: W3C Subresource Integrity, `http://www.w3.org/TR/SRI/`, W3C Candidate Recommendation, November, 2015
37. W3C: Web Cryptography API, W3C Candidate Recommendation, December, 2014