Transitioning to a Javascript voting client for remote online voting

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

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

Abstract: Voters in remote electronic voting systems typically cast their votes from their own devices, such as PCs and smartphones. The software executed at their devices in charge of performing the ballot presentation, navigation and most of the cryptographic operations required to protect the integrity and privacy of the ballot, is referred to as the *voting client*. The first voting clients were developed as Java Applets. However, the use of this technology has become relegated in front of web technologies such as Javascript, which provide a better multi-platform user experience. This is the reason why in 2013 Scytl decided it was imperative to develop a voting client purely based on Javascript. This industrial paper shows the implementation experiences and lessons learned during the development and deployment of Javascript voting clients for our remote electronic voting systems. The paper is complemented with a performance study of 1) the main cryptographic primitives used in voting clients and 2) the voting casting process of one of the voting clients used in a real election.

1 Introduction

In remote electronic voting systems, voters are usually allowed to cast their 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 fulfill 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 fulfill such security requirements, remote electronic voting systems use advanced cryptographic protocols (see for example (Adida, 2008), (Gjosteen, 2013), (Juels et al., 2010)). These protocols require to perform some 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. 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, being a source of security vulnerabilities, not even supported in most of mobile devices and with support recently removed from the most popular browsers. In 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 multi-platform 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.

Thus, the main contribution of this industrial paper is to show the implementation experiences and lessons learned, along with the whole process of developing and deploying Javascript voting clients used in several real elections. The paper is organized in seven sections: Section 2 describes a generic voting client and the cryptographic operations it may

Scytl
Innovating Democracy
SWISS POST

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 analyses the performance of cryptographic primitives and a voting client implemented in Javascript, and also explains some of the approaches followed to improve it; finally Section 7 presents the conclusions reached.

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 voting system is composed of two main components, 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 use a mix-net (Chaum and Pedersen, 1992) in order to shuffle and transform the encrypted votes prior to decryption.

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 components

The voting client comprises a graphical user interface and some underlying logics. These last ones can be divided in several components:

- **Authentication**: This component authenticates the voter in front of the voting system. The component can be adapted for integration with the customer infrastructure or totally replaced by a third-party authentication system.
- **Vote generation**: This component generates the ballot to be cast, i.e. encoding the voting options selected by the voter, encrypting and digitally signing the ballot containing them, and sending 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 components 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 depends on the specific voting protocol implemented each one providing different security properties under different assumptions (Gharadaghy and Volkamer, 2010), (Puiggalí et al.,). Despite this, a generic set of operations can be defined: 1) The voter authenticates to the system; 2) upon presentation of her ballot, she selects the voting options that represent her voting intent; 3) after the voter confirms her vote, the voting options are encoded and encrypted; 4) mathematical proofs of correct encryption are generated (if required by the protocol); 5) the encrypted voting options and the mathematical proofs are digitally signed; 6) the vote is cast to the remote server; 7) the server provides a confirmation receipt which is presented to the voter.

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:

- **Hash functions**: they are used for computing certificate fingerprinting and digital signatures. Although the standard for SHA-3 has already been published (NIST, 2015), the previous standard SHA-2 (NIST, 2012) (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 algorithm with

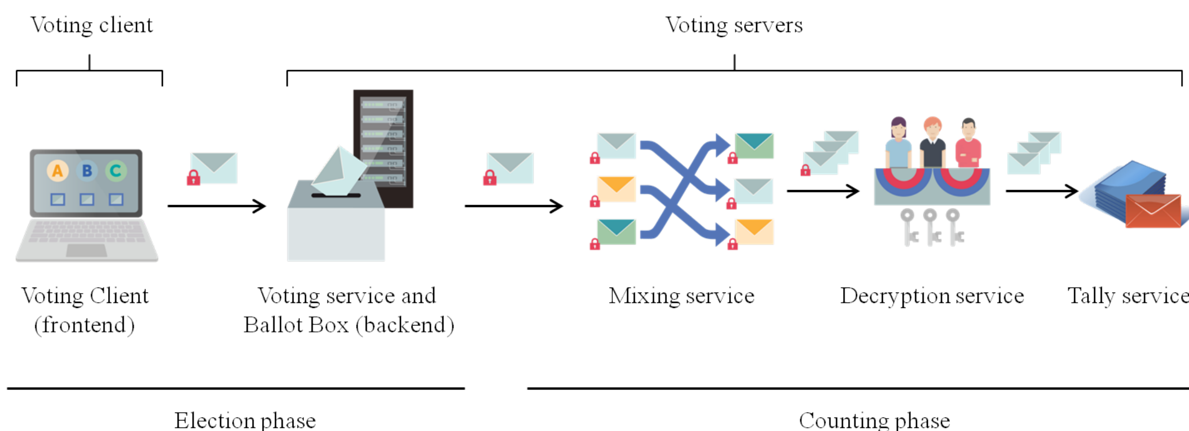

Figure 1: Remote electronic voting system

the hash variant (*RSA Full Domain Hash signature scheme (RSA-FDH)* (Bellare and Rogaway, 1993)) has been considered.

- **Encryption algorithms**: they are used for encrypting the voting options. RSA and ElGamal encryption algorithms (Menezes et al., 1996), and the AES encryption algorithm (NIST, 2001), are considered for public key and symmetric key cryptography respectively.
- **Zero-Knowledge Proofs of Knowledge (ZKPK)**: they (Damgaard, 2010) are used to prove a certain statement without revealing any other information than the statement is true. For example the Schnorr Signature (Schnorr, 1991) can prove knowledge of the randomness used for encrypting a message, providing assurance of the originator of an encrypted vote, and preventing vote copying (Cortier and Smyth, 2011).
- **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 (RSA Laboratories, b) 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 (RSA Laboratories, a) 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 (RFC-5280, 2008) 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

3.1.1 Availability of cryptographic primitives

JavaScript does not implement the cryptographic functionalities required by the voting client. An analysis of existing third party libraries was done to determine which of the required functionalities were provided by each library. The following factors were considered: functionality provided, in order to know if the library provided exactly what was needed, or if it required some modification; confidence level, the popularity of the library and the maintenance of the code; license, to analyse if the license terms were compatible with the license of the voting client application. In our particular context we preferred BSD, MIT or LGPL licensed libraries.

The results of the analysis, updated on April 2015, are shown on Table 1. The conclusion was the Forge[1] library is one of the best alternatives to implement a voting client. It provides functions for parsing and opening PKCS#12 containers, as well as managing digital certificates and public key cryptography. Besides this, it has a strong developer support and updates are made constantly. Both SJCL[2] (Stark et al., 2009) and jsbn[3] are also broadly used in other software. Specifically, jsbn is used in other of the ana-

[1]http://digitalbazaar.com/forge
[2]http://crypto.stanford.edu/sjcl
[3]http://www-cs-students.stanford.edu/~tjw/jsbn

Functionality	SJCL	Forge	jsbn	jsrsasign	CryptoJS
SHA-256	√	√	X	√	√
RSA signature with SHA-256	X	√	X	√	X
RSA encryption	X	√	√	X	X
ElGamal encryption (over Zp)	X	X	X	X	X
AES encryption	√	√	X	X	√
Schnorr Signature or ZKPKs	X	X	X	X	X
BigInteger support	√	√	√	√	X
PRNG	√	√	√	X	X
Parsing PKCS#12 containers	X	√	X	X	X
Parsing X.509 certificates	X	√	X	√	X
PBKDF2	√	√	X	√	√
Confidence level	Maintained	Maintained	No updates	Few updates	Few updates
License	BSD, GPL-2.0	BSD 3-Clause, GPL-2.0	BSD	MIT	BSD

Table 1: Cryptographic functionalities provided by third party libraries

lyzed libraries to provide BigInteger support (Forge, jrsasign[4]), and in the Helios voting system (Adida, 2008). SJCL is a library developed by highly recognized 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[5] 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 4.1.1).

3.1.2 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()* (ECMAScript, 2011), its implementation is not considered cryptographically secure (Klein, 2008). Alternatively, there is a W3C standardization initiative for a JavaScript API for performing basic cryptographic operations (W3C, b). This includes a Pseudo Random Number Generator (PRNG) suitable for cryptographic uses, *window.crypto.getRandomValues()*, which uses the browser's host system entropy, but is only implemented in recent versions of the browsers and not for all mobile devices' browsers[6].

Some of the cryptographic libraries analized have their own PRNG implementations. Specifically, SJCL and Forge implement the Fortuna PRNG by Schneier and Ferguson (Ferguson and Schneier, 2003). 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 (Stark et al., 2009). 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).

[4]http://kjur.github.com/jsrsasign
[5]http://code.google.com/p/crypto-js
[6]https://developer.mozilla.org/en-US/docs/DOM/window.crypto.getRandomValues

3.2 Security

3.2.1 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 dep-

recated, solution[7], 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* (W3C, a) 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 (Karapanos et al., 2016).

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

3.2.2 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 fulfill 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 (Halderman and Teague, 2015) that a team of researchers discovered in the Javascript voting client that we implemented for the State General Elections 2015 of New South Wales[9]. 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* (W3C, a) 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 (Adida, 2008) 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 optimizations had to be performed at the cryptographic protocol level (see Sections 4.1.1 and 4.3) to provide reasonable voting times.

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 con-

[7] http://www.mozilla.org/projects/security/components/signed-scripts.html
[8] http://aide.sourceforge.net
[9] http://www.vote.nsw.gov.au

sequence, a cryptographic library containing a large amount of primitives was developed:

4.1.1 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 digital signature, AES 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.

4.1.2 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 (Adida, 2008), (Gjosteen, 2013), (Galindo et al., 2015). 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 optimization (Gennaro, 2005; van Oorschot and Wiener, 1996) 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 (Koshiba and Kurosawa, 2004).
- **ElGamal encryption with multiple keys**: When the number of plaintexts to be encrypted is higher than one, an optimization 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 (Gjosteen, 2013; Kurosawa, 2002).

In order to maximize the code reuse and reduce the likelihood of errors, we used the Maurer framework (Maurer, 2009), which generalizes 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 initialized, 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 2) detailed in the following sections.

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*

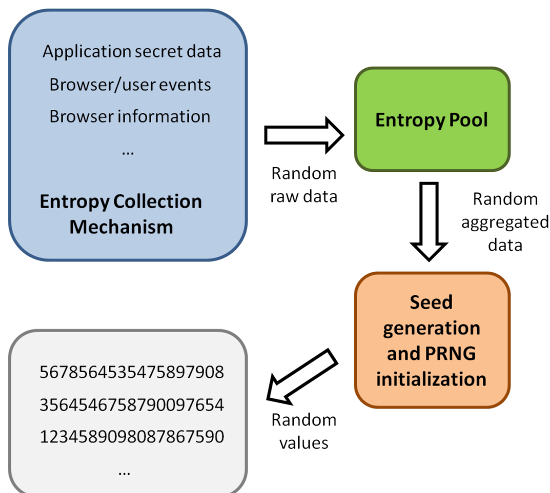

Figure 2: Diagram of the Pseudo-Random Number Generator

- 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 (Stark et al., 2009).

Seed generation and initialization 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 initializing 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 (Ferguson and Schneier, 2003) 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.

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 (Gjosteen, 2013) 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 (Galindo et al., 2015), 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.

Defense 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 exptected quality.

5.1 Testing tool

Dieharder (Brown et al., 2009) 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 (Marsaglia, 1996), as well as tests from the Statistical Test Suite (STS) (Rukhin et al., 2010) 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 initialized 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 (initialized with different seeds) generate uncorrelated random numbers.

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

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.

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 *trully 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 summarizes 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 distributed. More information about the null hypothesis and the p-values can be

found in (NIST, 2010) and the Dieharder manuals[10].

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. 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 operations, 100 executions for the PBKDF2 operation [11], and 10 executions for the rest of operations respectively. Furthermore, the PBKDF2 is setup with 1000 iterations and computes a key of length 256 bits. ElGamal and RSA algorithms are tested with 2048-bit keys.

Several conclusions can be extracted from the results. First, the performance is heavily influenced by the type of device, i.e. regular PC or mobile device, and browser. As expected, the regular PC performs faster since it is more powerful than a mobile device. There is also a huge difference between different browsers, e.g. Google Chrome and Firefox perform certain operations more than 6 times faster than Internet Explorer. The operating system does not have a strong impact on the results, e.g. similar results are obtained in Linux and Windows. Regarding the operation types, hashing is much faster than asymmetric encryption and decryption, as expected. The results also show a 5-7 times speedup in the optimized ElGamal encryption based on short exponents (see Section 4.1), reaching a speed closer to RSA.

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.
- **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.

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

[11] PBKDF2 time for iPhone 5s had a typographic mistake in the published paper that is currently fixed

Device	PC i5-2450M CPU 2,50GHz, 4GB RAM				LG G2	iPhone 5s
OS	Ubuntu 14.10	Windows 7			Android 4.4.2	iOS 8.2
Browser	Chrome 42	Chrome 42	Firefox 37	IE 11	Chrome 42	Safari
SHA-256	0,053	0,042	0,051	0,040	0,114	0,127
HMACwithSHA256	0,065	0,041	0,040	0,042	0,150	0,204
Generate PBKDF2 with SHA256	9,480	9,510	12,780	23,880	31,620	45,410
AES-128 Enc / Dec	0,031 / 0,017	0,031 / 0,015	0,045 / 0,024	0,179 / 0,008	0,197 / 0,097	0,146 / 0,072
ElGamal Enc / Dec (Normal Exp)	343 / 180	354 / 183	285 / 154	1.888 / 940	1.002 / 500	1.505 / 790
ElGamal Enc / Dec (Short Exp)	48 / 29	49 / 29	47 / 23	252 / 139	160 / 105	213 / 134
Schnorr Proof Gen / Ver (Short Exp)	24 / 47	24 / 48	21 / 40	137 / 288	68 / 135	108 / 213
RSA Enc / Dec	2 / 48	2 / 48	2 / 45	8 / 259	7 / 156	8 / 213

Table 3: Cryptographic operations performance (time in ms)

6.2.1 Neuchâtel e-voting protocol

The selected voting protocol is the one used in a recent election in Neuchâtel e-voting platform (Galindo et al., 2015) (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 summarized 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 4.1.1, 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 (Schnorr, 1991) identification protocol, proves the encrypted ballot is well-formed. The second, based on the Chaum-Pedersen (Chaum and Pedersen, 1992) 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 4.1.2 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.

6.2.2 Results obtained

The results (see Table 4) are aligned with the ones obtained for the cryptographic primitives, where the platforms chosen have a strong influence on them. 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.

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 plat-

Device	OS	Browser	Time
PC	Windows 7	IE10	22 sec
PC	Windows 7	IE11	5 sec
PC	Windows 7	Firefox 35	5 sec
PC	Windows 7	G. Chrome 39	5.5 sec
PC	Ubuntu	Firefox 35	5 sec
PC	Ubuntu	G. Chrome 39	6 sec
Mac	Mac OS 10.8	Firefox 35	5.5 sec
Mac	Mac OS 10.8	G. Chrome 39	5.5 sec
Mac	Mac OS 10.8	Safari	6 sec
HTC Desire 610	Android 4.4	G. Chrome 39	43 sec
Nokia Lumia 730	W. Phone 8.1	IE Mobile 11.0	29 sec
iPhone 5C	iOS 8.1.2	Safari 8.0	18 sec
S. Galaxy Tab3	Android 4.2.2	G. Chrome 39	23 sec
Enc. M. WT7-C-100	Windows 8.1	IE 11	14 sec
iPad Air	iOS 8.1.2	Safari 7.0	11 sec

Table 4: Vote casting performance in a HP Intel Core i5-3470 3.2 GHz PC, with Windows 7 Professional OS

forms. 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[12] were used if available. This 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 optimizations 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.

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

7 Conclusions

In this paper we have described our experience and lessons learnt on implementing a Javascript voting client in an industrial environment, focusing on the main challenges encountered, implementation details and performance results obtained.

In comparison to Java implementations, the overall security of Javascript voting clients is similar, whereas the user experience and interoperability is largely better since these clients are much lighter and multi-platform than the Java ones. A browser with Javascript support is the only usage requirement. The fact that there is no dependency with the Java Runtime Environment (JRE) has extensively reduced the usability and interoperability problems, as well as the exposure to critical security bugs, associated to the former Java implementations. Regarding the security, the only disadvantage is the Javascript implementation lacks the support for signing the code. However, 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. In addition, we also reported on the usage of the client in real elections, with a positive outcome.

Further work is being peformed to deal with the Javascript weaknesses, mostly the lack of code signed support. On the one hand, an intelligent application for remote code integrity validation service is being implemented. This application issues requests, which should not distinguishable from regular requests issued by real voters, to retrieve and validate the code served. On the other hand, the development of specific voting client apps for the most popular mobile platforms is studied, since these provide code integrity mechanisms for their applications.

REFERENCES

Adida, B. (2008). Helios: Web-based open-audit voting. In van Oorschot, P. C., editor, *USENIX Security Symposium*, pages 335–348. USENIX Association.

Bellare, M. and Rogaway, P. (1993). Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA. ACM.

Brown, R. G., Eddelbuettel, D., and Bauer, D. (2009). Dieharder: A random number test suite. *Duke University Physics Department*.

Chaum, D. and Pedersen, T. P. (1992). Wallet databases with observers. In Brickell, E. F., editor, *Advances in Cryptology - CRYPTO '92, 12th Annual Interna-*

tional Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer.

Cortier, V. and Smyth, B. (2011). Attacking and fixing Helios: An analysis of ballot secrecy. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011*, pages 297–311. IEEE Computer Society.

Damgaard, I. (2010). On σ-protocols. Cryptologic Protocol Theory, CPT 2010, v.2.

ECMAScript (2011). ECMAScript ® Language Specification 5.1 Edition.

Ferguson, N. and Schneier, B. (2003). *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 1 edition.

Galindo, D., Guasch, S., and Puiggalí, J. (2015). 2015 Neuchâtel's cast-as-intended verification mechanism. In Haenni, R., Koenig, R. E., and Wikstrom̈, D., editors, *E-Voting and Identity*, volume 9269 of *Lecture Notes in Computer Science*, pages 3–18. Springer International Publishing.

Gennaro, R. (2005). An improved pseudo-random generator based on the discrete logarithm problem. *J. Cryptology*, 18(2):91–110.

Gharadaghy, R. and Volkamer, M. (2010). Verifiability in electronic voting - explanations for non security experts. In Krimmer, R. and Grimm, R., editors, *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*, volume 167 of *LNI*, pages 151–162. GI.

Gjosteen, K. (2013). The norwegian internet voting protocol. ePrint. eprint.iacr.org/2013/473.pdf.

Halderman, J. A. and Teague, V. (2015). The new south wales ivote system: Security failures and verification flaws in a live online election. In Haenni, R., Koenig, E. R., and Wikström, D., editors, *E-Voting and Identity: 5th International Conference, VoteID 2015, Bern, Switzerland, September 2-4, 2015 Proceedings*, pages 35–53. Springer International Publishing.

Juels, A., Catalano, D., and Jakobsson, M. (2010). Coercion-resistant electronic elections. In Chaum, D., Jakobsson, M., Rivest, R. L., Ryan, P. Y. A., Benaloh, J., Kutylowski, M., and Adida, B., editors, *Towards Trustworthy Elections, New Directions in Electronic Voting*, volume 6000 of *Lecture Notes in Computer Science*, pages 37–63. Springer.

Karapanos, N., Filios, A., Popa, R. A., and Capkun, S. (2016). Verena: End-to-end integrity protection for web applications. In *2016 IEEE Symposium on Security and Privacy (to appear)*, pages 895–913.

Klein, A. (2008). Temporary user tracking in major browsers and Cross-domain information leakage and attacks. September-November, 2008, Trusteer.

Koshiba, T. and Kurosawa, K. (2004). Short exponent diffie-hellman problems. In Bao, F., Deng, R. H., and Zhou, J., editors, *Public Key Cryptography - PKC 2004, 7th Int. Workshop on Theory and Practice in Public Key Cryptography*, volume 2947 of *Lecture Notes in Computer Science*, pages 173–186. Springer.

Kurosawa, K. (2002). Multi-recipient public-key encryption with shortened ciphertext. In Naccache, D. and Paillier, P., editors, *Public Key Cryptography, 5th Int. Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002*, volume 2274 of *Lecture Notes in Computer Science*, pages 48–63. Springer.

Marsaglia, G. (1996). Diehard: a battery of tests of randomness.

Maurer, U. (2009). Unifying zero-knowledge proofs of knowledge. In Preneel, B., editor, *Progress in Cryptology AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 272–286. Springer Berlin Heidelberg.

Menezes, A. J., Vanstone, S. A., and Oorschot, P. C. V. (1996). *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

NIST (2001). Federal Information Processing Standard (FIPS) 197, Advanced Encryption Standard (AES). Technical report, U.S. Department Of Commerce.

NIST (2010). A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications, NIST Special Publication 800-22rev1a. Technical report, U.S. Department Of Commerce.

NIST (2012). Federal Information Processing Standard (FIPS 180-4), Secure Hash Standard. Technical report, U.S. Department Of Commerce.

NIST (2015). Federal Information Processing Standard (FIPS) 202, SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical report, U.S. Department Of Commerce.

Puiggalí, J., Chóliz, J., and Guasch, S. Best practices in internet voting. In *NIST: Workshop on UOCAVA Remote Voting Systems. Washington DC, August 2010.*

RFC-5280 (2008). Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List Profile.

RSA Laboratories. PKCS #12: Personal Information Exchange Syntax Standard.

RSA Laboratories. PKCS #5: Password-Based Cryptography Standard.

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., and Iii, L. E. B. (2010). *NIST Special Publication 800-22 Rev 1a: Statistical test suite for random and pseudorandom number generators for cryptographic applications*.

Schnorr, C. P. (1991). Efficient signature generation by smart cards. *J. Cryptol.*, 4(3):161–174.

Stark, E., Hamburg, M., and Boneh, D. (2009). Symmetric cryptography in JavaScript. In *ACSAC*, pages 373–381. IEEE Computer Society.

van Oorschot, P. C. and Wiener, M. J. (1996). On diffie-hellman key agreement with short exponents. In Maurer, U. M., editor, *Advances in Cryptology - EUROCRYPT '96, Int.l Conf. on the Theory and Application of Cryptographic Techniques*, volume 1070 of *Lecture Notes in Computer Science*, pages 332–343. Springer.

W3C. W3C Subresource Integrity. W3C Candidate Recommendation, November, 2015.

W3C. Web Cryptography API. W3C Candidate Recommendation, December, 2014.