Elliptic Curve Cryptography in JavaScript

Laurie Haustenne, Quentin De Neyer, and Olivier Pereira*

Université catholique de Louvain
ICTEAM – Crypto Group
B-1348 Louvain-la-Neuve – Belgium

Abstract. We document our development of a library for elliptic curve cryptography in JavaScript. We discuss design choices and investigate optimizations at various levels, from integer multiplication and field selection to various fixed-based EC point multiplication techniques.
Relying on a small volume of public precomputed data, our code provides a speed-up of a factor 50 compared to previous existing implementations. We conclude with a discussion of the impact of our work on a concrete application: the Helios browser-based voting system.

1 Introduction

Current browsers offer fairly limited support for performing cryptographic operations on the client-side of web applications. The support of the TLS/SSL protocols enables secure client-server communications, but these protocols can only be useful in settings where the server is trusted by the client, and the implemented cryptographic libraries are not exposed for other uses by web applications.

There are numerous applications, however, in which it is not desirable to ask web application users to trust a server. E-voting is one of them: encrypting ballots on the client side using a key that does not allow the server to decrypt the vote content not only limits the trust that the voters need to place in the voting server, but also substantially decreases the incentives for an attacker to hack the voting server, since the server then only sees information that it cannot interpret. While e-voting was the initial motivation for our work, being able to run cryptographic protocols on the client-side also offers very interesting perspectives for many other web applications, e.g., browser synchronization [18] or auctions [6].

The JavaScript engine appears to be the most convenient choice for computing on the client side of web applications: a JavaScript engine is provided with all major browsers. The interest of a cryptographic library in JavaScript is however not limited to browsers, as JavaScript is also available and increasingly used in other contexts in which cryptography is useful: one can think for instance about documents such as PDF or OpenOffice files, but also about server-side environments like Node.js.

* Olivier Pereira is a Research Associate of the Belgian Funds for Scientific Research F.R.S.-FNRS.

These various applications indicate that cryptographic libraries in JavaScript would be very useful, and it is therefore not surprising that various such libraries have been proposed already [4, 10, 20–22]. Even though some of these libraries offer some level of support for ECC, the design criteria of these libraries are essentially undocumented.

Running cryptographic operations in JavaScript in a browser presents constraints that are quite different from those appearing in classical cryptographic applications. On the one hand, despite tremendous improvements during the last two years, the performance of JavaScript code remains extremely low compared to optimized compiled code executed on the same computer. On the other hand, compared to other slow platforms like smart-cards, browsers offer an amount of memory that is larger by orders of magnitude. Such constraints motivated our independent study.

Our contributions. We present our development of elliptic curve cryptographic primitives in JavaScript, offering the first documented study on this topic. In particular:

- We compare several integer multiplication algorithms, determining when the grade-school multiplication technique becomes outperformed by asymptotically more efficient algorithms like the Karatsuba multiplication.
- We compare the performances of operations in various finite fields (binary, prime order, OEF).
- We define new NIST-style elliptic curves that are optimized for JavaScript implementation.
- We compare several fixed-base point multiplication algorithms, and determine which ones are the most efficient as a function of the number of points that one desires to store during precomputation.

Our implementation of EC point multiplication is more than 50 times faster than the most efficient stable one [22], offering comparable security levels. We stress however that this implementation does not rely on precomputation, while we rely on a small volume of public precomputed data.

The remaining parts of this document are organized as follows. In Section 2, we document our experiences with integer multiplication and various field operations, leading to the selection of new curves. In Section 3, we discuss various point multiplication strategies. We then discuss applications of our work in the context of voting protocols in Section 4, and conclude.

2 Field operations

We discuss the results of our investigation of arithmetic in prime fields. Our investigation however also involved binary and optimal extension fields, but they showed to be less efficient for our purpose. A summary of our results for these other types of fields is provided at the end of this section, and a detailed account is available in a separate report [12].

2.1 Big integer representation

JavaScript does not offer any support for the manipulation of big integers: one single numeric literal exists [14], and numbers are represented as IEEE-754 doubles.

In order to tackle this limitation, various strategies have been adopted. One possibility is to use the LiveConnect feature of web browsers that enables JavaScript to intercommunicate with a Java Virtual Machine: support for big integers and for basic operation on these integers is then provided by the JVM. This is the approach that was adopted in the Helios voting system for instance [1, 2, 8]: Helios performs big integer manipulations like modular exponentiation through LiveConnect, but all higher level algorithms (ElGamal, ...) are implemented in JavaScript directly. While this allows taking benefit of the JVM, this approach is also fairly limited in terms of algorithmic efficiency since only basic modular exponentiation is available: more efficient algorithms, for fixed-based exponentiation or multi-exponentiation for instance, are therefore not used.

Another approach, which became practical very recently due to the tremendous performance improvements of the JavaScript engines available in the major browsers, is to develop a pure JavaScript big integer library (educational implementations of such libraries have however been available for quite a long time). This is the approach we want to adopt here, as it removes the dependence of any external browser plug-in.

We take as our starting point the JSBN library by Tom Wu [22], which is, to the best of our knowledge, the most advanced big integer JavaScript library. In this library, big integers are stored as arrays of smaller integers, the length of which depends on the detected browser. Indeed, while JavaScript exposes signed 32 bits integers, considerable slowdowns appear when one computes with integers that come close to these 32 bits, as demonstrated in Table 1. Our experiments show that using arrays of 28 bit integers provides the most efficient results that are usable on the major browsers. For these measurements, we used an average netbook: Intel Core 2 Solo processor SU3500 (1.4 GHz) running Windows Vista. The browser version were as follows: FFX: Mozilla FireFox 4.0.1; IE: Internet Explorer 9.0.1; CHR: Google Chrome 11.0.696.71; SAF: Safari 5.0.5.

Table 1. Timings for multiplication in μs

	FFX	IE	CHR	SAF
28 bit words	5.3	7.6	3.2	8.2
30 bit words	13	16	4.3	12

As a result, in order to be able to exploit the integer representation in the choice of the field in which we compute, we decided to only use the 28 bits representation instead of having an adaptive integer representation according to the browser type.

2.2 Integer Multiplication

JSBN uses long (or grade-school) multiplication. It was not clear however whether performance improvements could come from using asymptotically more efficient algorithms. Therefore, we implemented the classical Karatsuba algorithm [15], which allows moving from $\mathcal{O}(n^2)$ complexity to approximately $\mathcal{O}(n^{1.585})$ complexity.

We provide an typical depiction of our experiments results in Figure 1, based on the Safari browser. As can be observed on this picture, Karatsuba multiplication becomes efficient for integers that are more than 1300 bits long. This bound is however strongly dependent of the browser that is used: on Firefox 3.6.23, the switch happens for 600 bit integers, while it happens only for integers around 1800 bit long on Chrome 14 (on the same Ubuntu laptop).

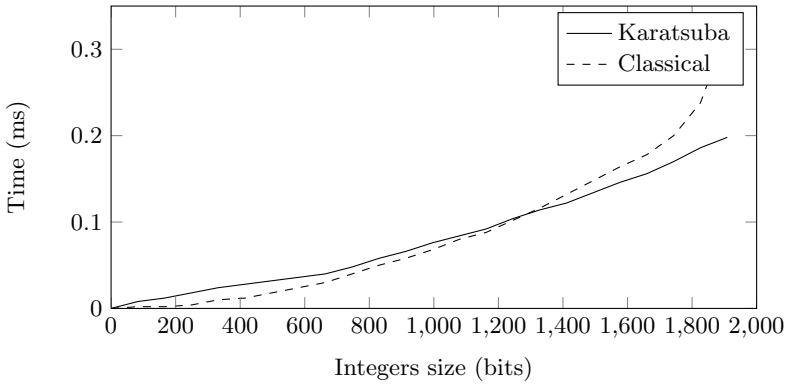

Fig. 1. Karatsuba multiplication becomes efficient around 1300 bit integers on our netbook running Safari.

While these integer lengths remain considerably longer than the integers we will manipulate for elliptic curve operations, this supports the adoption of Karatsuba multiplication (or of variants of it, e.g., Knuth or Toom-Cook) if one wishes to perform operations on larger integer. This might happen for cryptographic protocols that rely on the hardness of factoring, e.g., RSA encryption which was part of the motivations for the JSBN library, or Paillier encryption, but also if one desires to work in subgroups of $\mathbb{Z}_p^*$ for instance.

Nevertheless, for our purpose, we adopted standard grade-school multiplication. It would be interesting to see whether its efficiency could be further improved by using scanning techniques such as those proposed in [13] for instance.

2.3 Modular reductions

While computing in a prime field $\mathbb{F}_p$, reduction modulo p is a common and potentially expensive operation. In order to mitigate the cost of modular reductions, various ECC standards recommend using specially chosen primes that facilitate those reductions. For instance, the NIST prime p_{224} is equal to $2^{224}-2^{96}+1$ [19], in which we can observe that both 224 and 96 are multiples of 32, an expected word size for most implementations.

This 32-bit oriented choice is however clearly not optimal in our case, since our computation is based on 28-bit words. As a result, we looked for similar pseudo-Mersenne primes and found that $p_{224}^{28} = 2^{224} + 2^{140} + 2^{56} + 1$ is the prime integer with the fewest non-zero coefficients b_i in the set of integers of the form $2^{224} + b_7 2^{196} + b_6 2^{168} + b_5 2^{140} + b_4 2^{112} + b_3 2^{84} + b_2 2^{56} + b_1 2^{28} + b_0$ with $b_i \in \{-1, 0, 1\}$.

The JSBN library does not take into account the specific structure of the modulus when it performs reduction, and therefore does not exhibit any performance change when using p_{224}^{28}instead of p_{224}. Substantial changes appear, though, when using a specific modular reduction function, tailored for p_{224}^{28}.

The resulting performance of the prime field operations is given in Table 2, in which all timings include modular reduction. The squaring and inversion implementations are those from the JSBN library (except for the reductions), that is, the squaring is based on [17, Algorithm 4.16], and the inversion on [17, Algorithm 4.61].

Table 2. Timings for modular prime field operations in μs

	FFX	IE	CHR	SAF
addition	0.28	0.34	0.13	0.41
multiplication	**5.9**	**7.7**	**3.4**	**10**
squaring	4.9	6.2	3	8.5
inversion	900	1050	550	1100

This table shows fairly important discrepancies between the browsers. These values can however change substantially with browser updates. As expected, the inversion operation is by far the most expensive.

2.4 Result outline in other fields

Binary field arithmetic is typically slower than prime field arithmetic in software since integer multiplication is directly provided by processors and more efficient than repeated bitwise operations. As a result, multiplication showed to be on average 10 times slower on binary fields than on prime fields.

We also investigated optimal extension fields (OEF) [3]. These provide performances that are slightly slower than those of prime field arithmetic, except for inversion which is a bit more than 10 times faster.

Our efficiency measurements are summarized on a logarithmic scale in Figure 2 and detailed in [12]. The relative performance of operations in these fields is essentially in line with traditional results appearing in the literature for software implementation [11].

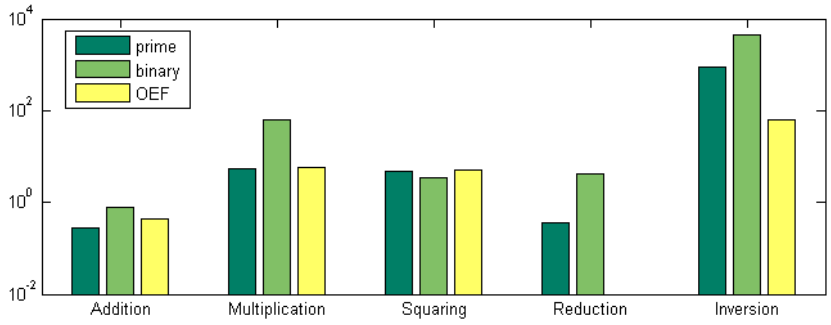

Fig. 2. Timing comparison of different field operations, in μs.

Since the main benefit of OEF, i.e., inversion, is not of interest for our applications, and since elliptic curves on OEF remain more experimental (they do not appear in the main ECC standards), we decided to adopt the prime field $\mathbb{F}_p$with our specially chosen prime $p_{224}^{28}$for the rest of our work.

3 Curve selection and operations

3.1 Curve selection

We selected a NIST-style pseudo-random curve [19] for p_{224}^{28}, that is, a curve of the form $E : y^2 = x^3 - 3x + b \bmod p_{224}^{28}$ of prime order n with base point (G_x, G_y). Our curve has the following parameters:

- $b = 136751745599456912706600915727146868999582204104477509956729818029 66$
- $n = 269599466671506397946670164808162043526395452929338422288298882185 79$
- $G_x = 150222183262519222405290909453932574140139625858373800570025968010 53$
- $G_y = 240399391475935753649984392771032630769178137930178136803571063783 07$

3.2 Choice of coordinates

The choice of a specific point representation has a substantial impact on the efficiency of point addition and doubling operations. Following the analysis provided by Hankerson et al. [11, Table 3.3], we decided to store points in affine coordinates, which is also the most efficient from a memory point of view, and to keep intermediate computation results in Jacobian coordinates. Using the algorithms from [11], the performance of point addition and doubling appear in Table 3. Investigating more recent techniques (e.g., those described in [5]), would certainly provide new improvements.

3.3 Point multiplication

The cryptographic protocols we consider involve a potentially large number of point multiplications, but only with a very small number of fixed bases (2 for ElGamal encryption for instance). Since a fairly large amount of memory is available in browsers for precomputation, exploring fixed-base point multiplication algorithms is particularly promising.

These algorithms could be used in two ways: either the browser performs precomputation himself and uses it later, or the precomputation is performed on the server side and provided to the browser as part of the web application (it could be certified and provided as part of the public key for instance).

We decided to adopt the second option, as requiring the browser to download a few extra kilobytes of public information is not an issue in our context. To fix the ideas, we decided to allow a volume of precomputed data of around 50 kB per base point, which corresponds to the volume of a small photograph. As we will see, a 10 times smaller volume of precomputed data already provides a very substantial acceleration, and nothing prevents to enable browser-based precomputation in bandwidth constrained environments (though different algorithmic choices should probably be made in that case).

We then explored various fixed point multiplication techniques, surveyed in [11] for instance: fixed-base windowing [7] based on standard and NAF representation and comb methods based on one or two precomputation tables (these methods are also detailed in [12]).

The relative complexity of these fixed point multiplication techniques is described in Figure 3, where the point doubling/addition ratio comes from our measurements of Table 3. We can observe that the two windowing techniques do not provide any extra benefit when more than 70 points are stored, while the two comb methods keep improving, the one based on two tables (comb2) being the most efficient. Generalization of the comb approach to more tables were also explored [16], but do not provide any improvement for the data volumes we have in mind.

Our limit of 50 kB of storage allows us to exploit more than 500 precomputed points. In this case, the complexity of a point multiplication is slightly lower than 50 point doubling operations. We observe that, by decreasing the number of stored points by a factor 10, the point multiplication complexity increases by a factor less than two, which might still be convenient if one desires to decrease the volume of precomputed data.

3.4 Point multiplication efficiency

The performance of our point operations is given in Table 3, based on the same computer and browser versions as before.

As before, those results are quite sensitive to the browser and computer that are used. For instance:

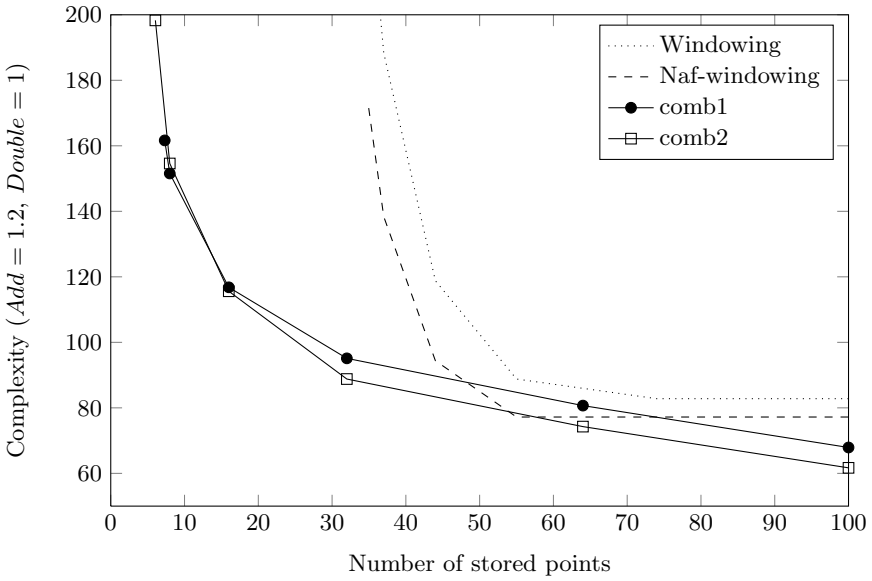

Fig. 3. Complexity of point multiplication as a function of the number of stored points. Windowing methods reach a minimum around 60 points, while comb methods keep improving.

Table 3. Timings for EC point operations in μs

	FFX	IE	CHR	SAF
addition	83	95	55	120
doubling	73	81	49	104
multiplication	**3300**	**3300**	**1900**	**4200**

- on a recent laptop (Intel Core i7-640M Processor at 2.8GHz) and using Chrome 14, a point multiplication operation takes $550\mu s$, which is already almost 4 times faster than the time reported for Chrome in Table 3,
- on an iPad 2, a point multiplication takes $14100\mu s$.[1]

Our implementation can also be compared to the one provided in the JSBN library [22], which is based on standard NIST curves, uses NAF point multiplication, and does not use precomputation. The point multiplication that took $550\mu s$ on the recent laptop mentioned above takes then around $30000\mu s$ with the JSBN implementation, presenting a slowdown of a factor 54 for the same security level. This gain comes from the various changes we made compared to the JSBN implementation: optimized modulus choice, specific modular reduction algorithm, choice of point representation, and precomputation.

[1] We thank Benoît Dumoulin for taking this measurement.

4 Application to e-voting

One possible use context for our ECC library is the Helios open-audit voting system [1, 2, 8], which has been used with two different cryptographic protocols on the client side:

1. The commonly deployed version, proposed in [2], is based on homomorphic tallying and uses a variant of the CGS protocol of Cramer et al. [9].
2. For some elections, mixnet-based tallying has also been used [8].

The homomorphic tallying approach enables a very simple election workflow, where the work of the election trustees is minimal: they only need to decrypt the election outcome, which is even cheaper than preparing a ballot. However, the ballot preparation procedure is fairly expensive for the voter, as it requires the equivalent of 6 point multiplications per candidate.

This computational complexity was the actual motivation for the adoption of a mixnet-based approach, when an election involving around 250 candidates was organized: adopting mixnets reduced the amount of computation to the equivalent of 5 point multiplications per ballot, but implied a substantially more complicated tallying procedure, including the setup of mix servers and requiring the trustees to decrypt all mixed ballots individually.

The fixed point multiplication techniques we explored in the previous section are particularly suitable for the CGS protocol. Indeed, all point multiplications are performed with respect to only two bases: a public group generator and an ElGamal public key which is made of a single point.

Using the Chrome browser on the average netbook described with our previous measurements, the time required to perform 1500 point multiplications when preparing a ballot for 250 candidates would be around 3 seconds, which is quite usable. It is not even necessary to require the voter to wait during those 3 seconds, as all point multiplications can be made independent of the voter choices, which can be encoded through point additions performed at the end of the ballot preparation procedure. The point multiplication operations can then be performed in separated worker threads while the voter performs his choices.

So, the library we presented in this paper provides an answer to the efficiency concern in Helios for elections involving a large number of candidates, and is expected to substantially increase the proportion of elections that can benefit from the simplicity of homomorphic tallying procedures.

5 Conclusion

Starting from the work of Tom Wu in the JSBN library for the support of big interger operations in JavaScript, we explored various strategies for the implementation of elliptic curve cryptography in pure JavaScript. Our resulting implementation, relying on a limited amount of precomputed data, offers a speedup of a factor 50 compared to the one proposed in the JSBN library. The efficiency of our implementation opens the way of substantial improvements in various

JavaScript applications, and we discussed the Helios voting system as an example.

There are a number of directions that remain open for further research.

- We concentrated our effort on NIST-type elliptic curves. It would be very interesting to explore whether other curve families would provide better results.
- Our library assumes that the precomputed data for fixed point multiplication are provided by an external application server. Including the cost of precomputation in the choice of the point multiplication technique would be another very interesting direction.

The adoption of our cryptographic library for real world applications remains currently limited by the lack of availability of secure randomness in JavaScript. Some efforts were already realized [4, 21], based on variants of the Fortuna design for entropy accumulation. More recently, since version 11, the Chrome browser exposes secure randomness through a new window.crypto.getRandomValues API, which provides a much more convenient and reliable solution. We hope to see secure randomness become available in other browsers within a near future.

Acknowledgments

We would like to thank Nicolas Veyrat-Charvillon for his support and the anonymous LC 2011 referees for their useful comments.

References

1. Ben Adida. Helios: web-based open-audit voting. In *Proceedings of the 17th USENIX Security Symposium*, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.
2. Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. Electing a University President Using Open-Audit Voting: Analysis of Real-World Use of Helios. In T. Moran D. Jefferson, J.L. Hall, editor, *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. Usenix, August 2009.
3. Daniel V. Bailey and Christof Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 472–485. Springer, 1998.
4. Marco Barulli and Giulio Cesare Solaroli. Clipperz. `http://www.clipperz.org`. Accessed on Oct 10, 2011.
5. Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *Advances in Cryptology - ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.
6. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, pages 325–343, Berlin, Heidelberg, 2009. Springer-Verlag.

7. Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David Bruce Wilson. Fast exponentiation with precomputation (extended abstract). In *Advances in Cryptology - EUROCRYPT '92*, volume 658 of *Lecture Notes in Computer Science*, pages 200–207. Springer, 1992.
8. Philippe Bulens, Damien Giry, and Olivier Pereira. Running mixnet-based elections with Helios. In H. Shacham and V. Teague, editors, *Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. Usenix, 2011.
9. Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In Walter Fumy, editor, *Advances in Cryptology - EUROCRYPT '97*, volume 1233 of *Lecture Notes in Computer Science*, pages 103–118. Springer, 1997.
10. Zhi Guan, Zhen Cao, Xuan Zhao, Ruichuan Chen, Zhong Chen, and Xianghao Nan. WebIBC: Identity based cryptography for client side security in web applications. In *28th IEEE International Conference on Distributed Computing Systems (ICDCS 2008)*, pages 689–696. IEEE Computer Society, 2008.
11. Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer-Verlag, 2004.
12. Laurie Haustenne and Quentin de Neyer. Elliptic curve cryptography in javascript with application for eVoting. Master's thesis, Universite catholique de Louvain, 2011.
13. Michael Hutter and Erich Wenger. Fast multi-precision multiplication for public-key cryptography on embedded microprocessors. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 459–474. Springer, 2011.
14. ECMA International. ECMAScript Language Specification – ECMA-262 rev. 5.1, 2011.
15. A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. In *Proceedings of the USSR Academy of Sciences*, volume 145, page 293–294, 1962.
16. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.
17. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, July 1999.
18. MozillaWiki. Weave cryptography developer overview. `https://wiki.mozilla.org/Labs/Weave/Developer/Crypto`, February 2010. Accessed on Oct 10, 2011.
19. NIST. FIPS PUB 186-3 – Digital Signature Standard (DSS). `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, 2009.
20. Dave Shapiro. RSA in JavaScript. `http://ohdave.com/rsa/`. Accessed on Oct 10, 2011.
21. Emily Stark, Michael Hamburg, and Dan Boneh. Symmetric cryptography in Javascript. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009*, pages 373–381. IEEE Computer Society, 2009.
22. Tom Wu. jsbn - BigIntegers and RSA in JavaScript. `http://www-cs-students.stanford.edu/~tjw/jsbn/`. Accessed on Oct 10, 2011.