JElliptic: Security assessment of elliptic curve cryptography using browsers and JavaScript

Bachelor project by

Solal Pirelli

Supervised by

Andrea Miele

Laboratory for Cryptologic Algorithms, LACAL

Ecole Polytechnique Fédérale de Lausanne, EPFL

January 9, 2015

Content

1 INTRODUCTION

In our modern, connected world, cryptography's role in protecting privacy is becoming more and more important. Many cryptographic systems are based on the fact that some problems seem to be inherently hard to solve, because they require computational power that nobody has, or will have in the future. It is very important to check that this assumption still holds from time to time, to make sure that our secure communications are not too easy to break.

Indeed, previous results such as (1) have shown that in some instances, systems believed to be secure were insecure because the computational power required was in fact obtainable. This has led to either the complete abandon of a system, or the change of some of its parameters.

Some attackers (such as governments, large-scale networks of virus-infected computers, or criminal organizations) do have lots of computational power, but this kind of power is often unavailable to security efforts, which makes it harder to assess the security of systems.

One approach to this problem is to take one specific instance of a system, with known values, and to distribute a computation that will, given enough power and time, find the private data that was protected, such as private information. If the computation can be done in relatively little time, then it must be assumed that attackers can also perform it, which means the system is insecure.

Elliptic curve cryptography (ECC) is a public-key cryptography system, based on the intractability of the *elliptic curve discrete logarithm problem* (ECDLP): given an elliptic curve over a finite field and two points g and h on the curve, find m such that $mg = h$.

ECC has only recently become mainstream in consumer-facing cryptography, such as TLS, which is used in secure Web communications (HTTPS). Most of its advantages come from the fact that keys can be much smaller, which lowers the computational power requirements from clients – something that is especially important in our mobile-first world – and reduces the transmission time of the key.

As expected from a widely used cryptosystem, there is no efficient method to solve the ECDLP. The best method currently known is *Pollard's rho method*. One key aspect of this method is that it can be trivially parallelized at virtually any scale, enabling the development of distributed solutions to run many instances of Pollard's rho at the same time on the same problem.

The goal of this project was to implement Pollard's rho as a website, in the JavaScript programming language, so that anyone could join a large-scale run of the algorithm using only their browser, which removes most of the hassle of participating in such an experiment: no need to download a native executable or borrow specialized hardware.

The most important challenge is that JavaScript was never intended for fast mathematical computations; its origins can be traced back to the wish to make web pages slightly dynamic, using very small scripts. Therefore, what made this project interesting was not to see if Pollard's rho could be implemented in JavaScript, but if it could run fast enough for practical purposes.

2 POLLARD'S RHO METHOD

2.1 DESCRIPTION

The main idea behind Pollard's rho method, originally described in a more general way in (2), is simple: perform a random walk over the elliptic curve, and wait for the walk to collide, i.e. encounter the same point twice.

More specifically, given an elliptic curve of order q and two points g and h on the curve, in order to find m such that $mg = h$, perform a walk given by the random coefficients u and v:

$$p = ug + vh$$

When the walk collides, for some $p = u_1 g + v_1 h = u_2 g + v_2 h$, then $m = \frac{u_1 - u_2}{v_2 - v_1} \bmod q$ solves the ECDLP, unless $v_1 = v_2$.

The *birthday paradox* tells us that the expected running time of Pollard's rho is $\sqrt{\frac{\pi q}{2}}$. While this may not seem very intimidating, one must remember that real-world q values are extremely large; massive computational power is necessary if one wishes to solve the ECDLP in reasonable time.

Such massive power needs not be one giant computer, or even one cluster of computers physically located in the same place. The trivial parallelization of this algorithm is to have any number of computers simultaneously walking on the curve, starting at different points, and have each computer report the points they find to a central server.

The "easy" way to implement Pollard's rho is by far not the best one, however.

2.2 DETERMINISM: R-ADDING WALKS

To enable many optimizations (some of which are explained below), every client needs to walk the curve in the exact same way, and they must do so in a deterministic manner; one cannot have every client generate random numbers – which is all but impossible for normal computers anyway – and try points that correspond to these numbers.

One way to do so is to define a *partition function* f over the elliptic curve, which maps every point in the curve to an integer in the space $[0, r-1]$ for some r. Then, generate an addition table containing r pairs of coefficients u_i and v_i, as well as the associated point $p_i = u_i g + v_i h$.

Given a point p, corresponding to the walk coefficients u and v, the next point is computed in two steps: First compute the index $i = f(p)$, and then add the pre-computed coefficients u_i and v_i to u and v respectively, as well as the pre-computed point p_i to p.

2.3 EFFICIENT PARALLELIZATION: DISTINGUISHED POINTS

Having every "client" computer report every point they find to a centralized server is feasible, but it slows down walks immensely. Since clients can easily perform (at the very least) thousands of walk steps per second, the time taken to send each point would slow down the algorithm to a crawl. Another concern is storage space on the server, which would very quickly be filled if all clients reported everything, even if they had some way to do it very quickly.

Therefore, the concept of *distinguished points* needs to be introduced. A distinguished point is a point that has some property, which can be anything as long as it is easily computable, and fairly rare. For instance, one can define a distinguished point as a point whose X coordinate ends with many 0s.

Clients will only report points if they are distinguished, drastically reducing the number of points that need to be sent back to the server. This does not mean that collisions are less likely to be found: as long as all clients use the same walk algorithm and that the algorithm is deterministic, two client whose walks collide at a non-distinguished point will keep on walking, and eventually both will report the next distinguished point on the walk.

2.4 Simultaneous inversion

Each step of a walk needs to add the "old" coefficients u, v with the pre-computed coefficients u_i, v_i, as well as add the "old" point p to the pre-computed point p_i. Adding the coefficients is "cheap" in terms of performance, but adding two points in an elliptic curve requires a few additions (which are cheap) as well as a modular inversion, which is expensive compared to the rest of the operations.

There is a trick to remedy to this problem: *Montgomery's simultaneous inversion*, as described in (3), which allows many inversions to be performed at the same time, using a few more multiplications but only one inversion.

Therefore, many walks are performed on the same machine, and their inversion steps are all done at once. Importantly, the walks are *not* ran in parallel on different processors or different processor cores; the walks are in fact done sequentially, in a loop.

2.5 Automorphisms

2.5.1 The negation map

Since random walks have to walk among every point in the curve, any way to efficiently reduce the number of points is welcome. One such way is to use an *automorphism* on the curve, i.e. a function that takes any point of the curve as input, and outputs a point among a subset of the curve.

All elliptic curves are horizontally symmetric, i.e. for each point (x, y) there is a point $(x, -y)$. Therefore, a suitable automorphism is a *negation map*: given a point $p = (x, y)$, compute $p_2 = (x, -y)$ and pick the one whose Y coordinate is the smallest, denoted as $\sim(p)$. This effectively reduces the search space by half, which means the expected running time is sped up by a factor of $\sqrt{2}$.

2.5.2 Cycles

Unfortunately, using the negation map is not as easy as it seems. It introduces the possibility of *cycles*: the walk will go through a series of points and eventually back to the first point in the series, which means it will keep looping in these points forever. This must, of course, be avoided.

The most common form of cycles is the 2-cycle, where the walk will go to one point, then to another, then back to the first, and so on. This is because the successor of p is $p + p_{f(p)}$ half the time, and $-p - p_{f(p)}$

half the time. In the latter case, with probability $\frac{1}{r}$ (where r is the number of pre-computed points), $f\left(-p-p_{f(p)}\right)=f(p)$ so the successor of $-p-p_{f(p)}$ is $-p-p_{f(p)}+p_{f(p)}=-p$, and $\sim(-p)=\sim(p)$. This happens with probability $\frac{1}{2r}$.

The easiest and simplest way to detect cycles is a "record and test" strategy: every α points, record β points and check if all of them are unique. If they are not, then the walk is in a cycle. The walk then has to escape the cycle, i.e. go to another point that will not lead back to the same cycle. This can be done in multiple ways, but many of them are subtly incorrect, as explained in (4). The safest way to escape cycles is to double the current point.

However, recording and testing points is relatively expensive, and cannot be done often enough compared to the frequency of 2-cycles. Another strategy has to be used to deal with them: whenever a new point is computed, immediately run f on it: if $f(\sim(p))=f\left(\sim(p+p_{f(p)})\right)$, "discard" the point to avoid a cycle. Discarding a point means that the current point stays the same, and the next point is determined by $\sim\left(p+p_{f(p)+1}\right)$. If that point, too, has the same result when passed to f, then use $\sim(p+p_{f(p)+2})$, and so on. In the very unlikely case (with probability r^{-r}) where no point passes the test, double the current point.

Thus, the overall speedup from using the negation map is less than the factor of $\sqrt{2}$ that could be hoped for, since walks need to deal with cycles.

2.6 Performance considerations

The number of entries in the addition table determines the probability of entering cycles when using the negation map; therefore, it should be fairly large. However, in most implementations of Pollard's rho method, using a negation map that is too large causes a large amount of processor cache misses, which severely decreases performance.

To reap the full benefits of simultaneous inversion, many walks are needed, but having too many of them can also cause problems if they can't all fit in the cache.

3 POLLARD'S RHO IN JAVASCRIPT

3.1 JAVASCRIPT

3.1.1 Origins

> *The by-design purpose of JavaScript was to make the monkey dance when you moused over it.*[1]
>
> *-Eric Lippert, former member of the ECMA JavaScript committee*

JavaScript is a dynamically-typed scripting language, created in the 90s by Netscape as an easy-to-use programming language, to let users with little knowledge of programming write simple.

Since then, it has evolved into a language in which entire web applications are written, sometimes containing hundreds of thousands of lines of code. Language features have been added, as well as libraries that allow JavaScript to manipulate complex data such as audio files, and even access graphics cards directly to a certain extent.

While the original JavaScript engines were very simple interpreters with little care for performance, modern engines use increasingly complex techniques to ensure applications run as fast as possible; the JavaScript performance has become a selling point of browsers, even for non-technical consumers.

3.1.2 Limitations

Some unfortunate limitations from the original design of the language remain.

The most relevant one for this project is that JavaScript only has one number type, the IEEE-754 double-precision floating-point number. The usual 32- or 64-bit integers do not exist[2]. Therefore, all mathematical code dealing with integers – as is the case for elliptic curve cryptography – must be extremely careful to avoid entering the realm of floats. Since the mantissa size of IEEE-754 double-precision floating-point numbers is 53 bits, the maximum positive integer that can be represented is $2^{53} - 1$. This means that all numbers that might be multiplied with other numbers must remain lower than $\sqrt{2^{53} - 1}$, otherwise the results may be incorrect.

Arrays are another pain point of the language: JavaScript does not have arrays by their common definition of "continuous block of memory of fixed length". Instead, all arrays can also be used as stacks (with push/pop operations), queues (with shift/unshift) operations, lists (with add/remove/insert), and even key/value maps. Modern engines help with this problem by using heuristics to detect "conventional" usages of arrays and optimizing for them, but the situation is still far from optimal.

[1] http://programmers.stackexchange.com/a/221658

[2] This is not strictly true, as 32-bit integers are used as intermediate values for bitwise operators. However, the result of these operators are converted back to floating-point numbers.

3.1.3 Performance

There are currently four mainstream JavaScript engines: Chakra (in Microsoft's Internet Explorer), Nitro (in Apple's Safari), V8 (in Google's Chrome) and *Monkey (in Mozilla's Firefox). All of them attempt to respect the JavaScript specification as closely as possible, and to run scripts as fast as possible.

Like all complex software, they have bugs; this is problematic because code that runs in most browsers may be broken in another one, making its use impossible. Performance can also vary wildly between browsers, with some optimizations being only implemented on some engines, which means some code patterns can be very fast or very slow depending on which browser the code is run on.

While these problems may seem comparable to the competition of compilers in languages such as C++, in the case of JavaScript, the code writer is not in control of which compiler is used. Picking the best one for a given piece of code is therefore impossible, and asking the user to please change their browser is not an option either: after all, the goal of this project is to increase the participation to large-scale distributed algorithm runs by making it easy to participate, i.e. removing the need to download an application or change one's computer habits.

3.2 IMPLEMENTATION

3.2.1 The repository

The code is open-source, and can be found at https://github.com/SolalPirelli/JElliptic.

Benchmarks and tests are hosted at http://solalpirelli.github.io/JElliptic.

3.2.2 TypeScript

The code is written in TypeScript, a superset of JavaScript that compiles down to it. As its name implies, its main addition is to let programmers write type annotations to ensure variables are of the expected types, which is extremely useful when developing software. The language compiles down to JavaScript, so these types are not preserved at run-time, and JavaScript engines cannot benefit from the annotations.

3.2.3 Big integers

JavaScript can only deal with relatively small integers (as seen above), and reasonably-sized elliptic curves are far beyond their range, thus an implementation of "big integers", i.e. integers that can be arbitrarily large, is required.

The project uses a custom implementation, with two fields: a Boolean value for the sign, and an array of numbers that are base-94906265 digits. The choice of base may seem odd at a first glance: it is the largest integer that is smaller than $\sqrt{2^{53}-1}$, for the precision reasons explained above. Since algorithms on numbers usually have a computational complexity associated with the number of digits, a higher base is better as it results in less digits for the same numbers.

3.2.4 Web workers

Each client should run Pollard's rho as an infinite loop walking over the curve. However, this is problematic when running inside a Web browser because of mechanisms to prevent long-running scripts from hanging pages; in most cases, a script that runs for more than a few seconds is an indicator of buggy code, such as an unintended infinite loop.

The infinite loop is intended in our case, so the code has to opt-in to a special functionality called *Web workers*, which lets scripts run in a background thread for as long as they wish, as long as they adhere to some additional constraints.

Theoretically, this means that parallel instances of the algorithm can be run on multiple threads at the same time. However, one instance is probably enough for most people; volunteers are already giving up one processor core to run computations on, and most computers only have two or four cores.

3.2.5 Optimizations

JavaScript engines are very sensitive to differences that may seem small. Here are some examples:

- Initializing an array with the empty array literal [ ] is faster than initializing it using the $new\ Array(length)$ syntax, even though the former does not include the length.
- Object properties should always be set in the same order, i.e. if one point is initialized by first setting its X value and then its Y value, all points should do the same (there shouldn't be points initialized by setting their Y value first). Otherwise, optimizers will not realize these are the same kind of objects.
- New JavaScript features are usually slow to use, because they have not been optimized yet, even when they seem focused on performance.
  "Freezing" objects to guarantee they will never change slows down operations considerably. Using "typed arrays" that can e.g. only contain numbers is slower than regular arrays.
- Allocations are fairly expensive, as JavaScript is garbage collected. A trade-off has to be reached between code readability and performance; the big integers implemented in this project are immutable from an external point of view but do sometimes use mutable state internally for performance.

Since elliptic curves use numbers that are fairly small for big integers, using "smart" algorithms with lower computational complexities than conventional algorithms is not worth it, as these algorithms have hidden constant factors.

3.3 Performance

3.3.1 Absolute performance

On a fairly powerful desktop computer, running Pollard's rho on the 112-bit elliptic curve on which an ECDLP instance was solved in (5):

$$y^2 = x^3 + 4451685225093714772084598273548424 * x + 2061118396808653202902996166388514\ (mod\ 4451685225093714772084598273548427)$$

Google Chrome performs up to 7,000 walk steps per second in a single walk, and up to 30,000 walk steps per second when using 128 parallel walks, reaping the benefits of simultaneous inversion. Even more walks does not seem increase performance.

Unlike what was described in section 2.6, performance is not affected by the number of entries in the addition table. This suggests that details such as cache misses are too low-level to be relevant in the context of a high-level scripting language like JavaScript.

Cycle detection, required by the use of the negation map, also seems to have a negligible effect on performance, which means the effective speedup is virtually identical to the $\sqrt{2}$ one could hope for. Celebrating this achievement would be premature: what it really means is that while cycle detection is relatively slow, the entire code is also fairly slow, which means cycle detection doesn't have as much of an impact as it does on implementations where mathematical operations are extremely fast.

3.3.2 Across browsers

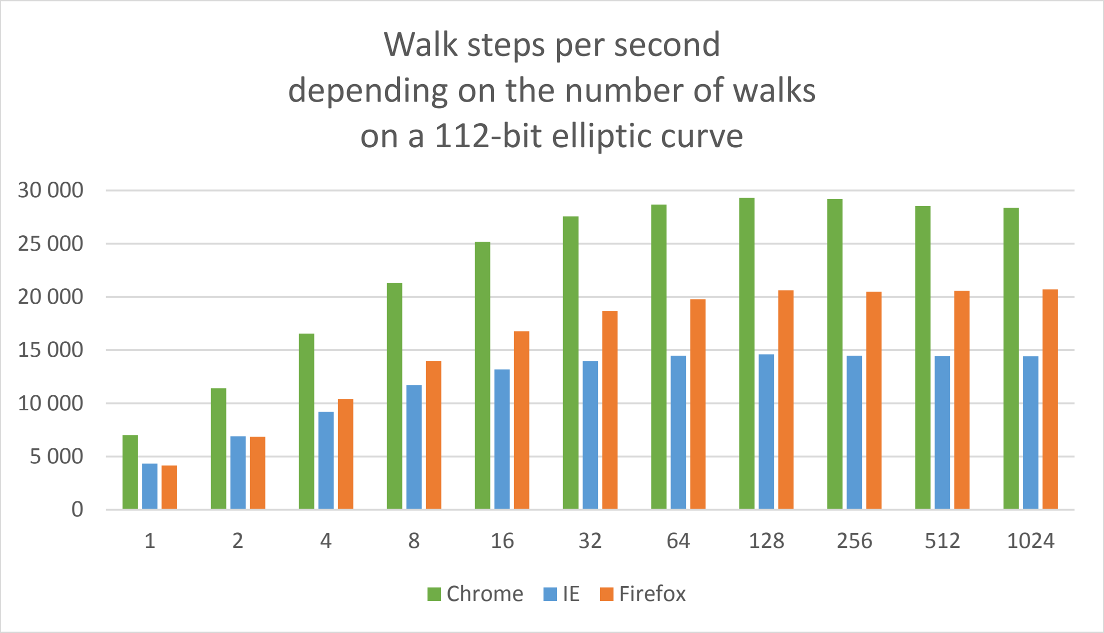

Figure 1: Benchmark results for Pollard's rho method in three different browsers. Environment: Windows 7, i7-4770.

This section uses the same elliptic curve as the previous one.

Chrome has clearly superior performance when compared to Internet Explorer and Firefox for this project, as seen in Figure 1. Internet Explorer performs marginally better than Firefox when using few walks, but trails significantly when the number of walks increase.

This is a clear demonstration of the effect of simultaneous inversion: Internet Explorer produces faster code for the inversion, but is slower at multiplications and additions, and therefore doesn't benefit as much from the simultaneous inversion as the other two. However, all three do benefit from it.

3.3.3 Comparison with native code

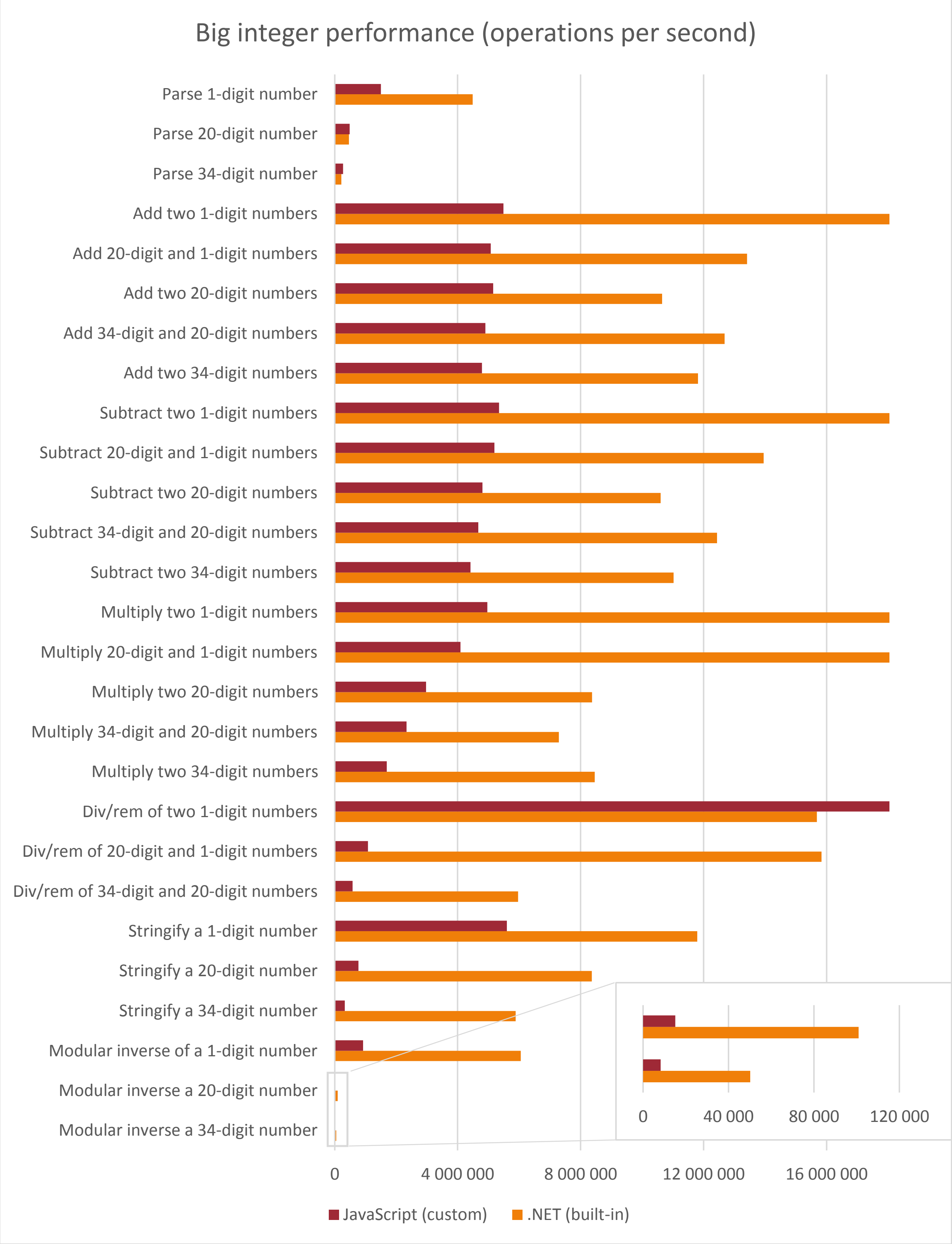

Figure 2: Big integer performance compared across platforms. Environment: Windows 7, i7-4770.

Since no fast implementation of Pollard’s rho method was easily available and modifiable to be benchmarked on an x86 platform, the JavaScript big integer library that was created for this project was benchmarked instead; the performance of Pollard’s rho can be extrapolated from it.

Figure 2 shows the comparison between the custom JavaScript big integer library and the one built into the .NET Framework (used by languages such as C#), which provides a good comparison point as it is a native big integer library tuned for speed, but *not* optimized for extremely large numbers (in the thousands of digits).

Most operations are about three times slower in the JavaScript library than in the .NET one; division is about ten times slower and modular inversion (which is made up of other basic operations) is six times slower.

Therefore, one can expect the JavaScript implementation to be about an order of magnitude slower than a native implementation, or perhaps by two orders when accounting for possible native optimizations such as SIMD.

3.3.4 Comparison with efforts on other platforms

In (5), Pollard’s rho was implemented on a cluster of PlayStation 3 (PS3) consoles. Each console computed an average of five points distinguished by a 24-bit mask every two seconds, i.e. $2^{24} * \frac{5}{2} \cong 4.2 * 10^7$ points per second, on the same curve as the one used for benchmarking this project.

PS3s are now fairly old: they have been superseded by a newer model, and are now less expensive than a powerful x86 CPU[3]. The x86 architecture does not seem to compete well with more specialized architectures in the case of elliptic curve cryptography.

[3] On the 8th of January 2014, a PS3 could be found for about 200 CHF, while an Intel i7-4770 CPU, used for some of the comparisons in this report, cost about 300 CHF.

4 CONCLUSION

The project's goal was to see whether JavaScript is fast enough to make distributed efforts at solving ECDLP instances easier by allowing volunteers to run Pollard's rho in their browser, without having to download any software: simply opening a link and clicking a button.

Compared to native code on the same consumer-grade hardware, the results are fairly promising. However, a comparison with more specialized hardware, such as the PS3s used in (5), casts doubt on the ability of large-scale efforts using JavaScript to successfully solve even relatively small instances of ECDLPs. Indeed, one must remember that real-world curves used in e.g. TLS have exponentially many more points than the "small" curves used for benchmarking in this paper.

The flip side of the coin is that current systems using ECC are most likely out of the reach of attackers such as large botnets made up of infected computers.

All in all, it seems that the resources needed for a distributed attempt at solving an ECDLP are too much compared to the resources needed to solve an ECDLP in a laboratory using specialized hardware.

5 References

1. **Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, et al.** *Factorization of a 768-bit RSA modulus.* 2010.

2. **Pollard, John M.** *Monte Carlo methods for index computation (mod p).* 1978.

3. **Montgomery, Peter L.** *Speeding the Pollard and elliptic curve methods of factorization.* 1987.

4. **Joppe W. Bos, Thorsten Kleinjung, and Arjen K. Lenstra.** *On the Use of the Negation Map in the Pollard Rho Method.* 2010.

5. **Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery.** *Solving a 112-bit Prime Elliptic Curve Discrete Logarithm Problem on Game Consoles using Sloppy Reduction.* 2012.