Ed448-Goldilocks, a new elliptic curve

Mike Hamburg*

Abstract

Many papers have proposed elliptic curves which are faster and easier to implement than the NIST prime-order curves. Most of these curves have had fields of size around 2^{256}, and thus security estimates of around 128 bits. Recently there has been interest in a stronger curve, prompting designs such as Curve41417 and Microsoft's pseudo-Mersenne-prime curves.

Here I report on the design of another strong curve, called Ed448-Goldilocks. Implementations of this curve can perform very well for its security level on many architectures. As of this writing, this curve is favored by IRTF CFRG for inclusion in future versions of TLS along with Curve25519.

1 Introduction

Since Edwards' discovery of a new elliptic curve form in 2007 [11], implementors have produced a steady stream of implementations in this form. Edwards curves tend to be easier to implement securely than the previously used curve shapes, because they support complete addition formulas. These formulas do not have exceptional cases which would lead to a division by zero [3]. Edwards curves are also faster, and have marginally simpler formulas as well. However, curves in Edwards or twisted Edwards form always have a cofactor divisible by 4, so prime-order curves such as the NIST curves cannot be put in this form.

Until recently, most Edwards curve implementations have been for fields of size around 2^{256} [2, 3, 7, 8, 12, 15, 17, 18, 22], making them comparable in security to NIST-P256 and AES-128 standards. However, recently several authors have proposed elliptic curves with field sizes ranging roughly from 336 bits to 521 bits [1, 4, 9, 21].

Here I detail the design of an Edwards curve with a 448-bit field. I hope that this curve will provide enough security to satisfy conservative users, but still be fast enough for those who are performance-conscious. It would therefore be useful as a more conservative supplement to Curve25519 and Ed25519.

As of early June 2015, the Internet Research Task Force Crypto Forum Research Group (IRTF CFRG) favors Ed448-Goldilocks for inclusion in future TLS standards [20].

2 Security rationale

Before going on, it is important to consider why a stronger elliptic curve would be desirable. In particular, why would anyone need a curve stronger than the existing 256-bit-field curves? These curves are said to require about as much work to break as AES-128 (e.g. Curve25519 has work factor $W := \frac{1}{2}\log q \approx 126$), but in strong attack models they may require more work than AES. Depending on the mode, symmetric encryption may be susceptible to multiple-target attacks, which

*Rambus Cryptography Research.

could allow an attacker to recover the first key of n targets in time approximately $2^{128}/(n+1)$. For elliptic curves, batch attacks do not speed up the recovery of the first key, only of subsequent ones.

What sorts of attacks might break an elliptic curve with a conjectured security level near 128 bits? Here are several possibilities:

- An attacker could use brute force. This is unlikely to be feasible for several decades at least, but designers might be concerned about it for very long-term security.
- An attacker might build a quantum computer capable of running Shor's algorithm. This would break every elliptic curve cryptosystem which could fit in that computer's memory, so a larger curve would not be helpful.
- A mathematical breakthrough might render all elliptic curve cryptography weak, or at least much weaker than expected. Depending on the breakthrough, a larger curve might resist attack due to its size, or it might fall along with the smaller ones.
- A breakthrough might break only curves with special properties, such as complex multiplication or a certain field shape. Or it might break all curves which do not have those properties.
- A protocol might have a loose security bound, and might allow an attacker to break it with only a tiny fraction of the work of solving the discrete log problem. This might enable an attack on curves which were previously out of reach, but only in that protocol.
- Security bugs or side channels might compromise an implementation. The defense against this is simplicity, not field size.
- Security architects might want to over-engineer a system, for marketing or just for extra confidence. Or having already done this, they might want to migrate from the NIST or Brainpool elliptic curves to an Edwards curve, without weakening their design.

Some of these issues can be mitigated by using a curve which is slightly bigger, such as Scott's curve modulo $2^{336}-3$ [21]. Others favor a curve which is significantly larger, but it is difficult to evaluate exactly how large. If a significantly larger curve is chosen, the usual work factor estimate means almost nothing, since any attempt to attack such a curve would require a significant breakthrough to obsolete that estimate.[1]

I decided to design a curve in the "significantly larger" or "overkill" level. The currently popular curves in this space include NIST [19] and Brainpool curves [10] at 384, 512 and 521 bits, so I aimed for something in this range. Since it is always possible to use a larger curve at the cost of worse performance, I aimed to find the best trade-off between performance and field size. I followed the `safecurves.cr.yp.to` policies for curve generation to avoid known mistakes and to demonstrate that nothing is up my sleeve.

I propose Ed448-Goldilocks as a single high-strength curve for new designs. It is tempting to suggest a 384- and a 521-bit curve to match the existing NIST and Brainpool sizes. But there is really no need to have an "overkill" curve and a "more overkill" curve: users will just use one and not the other.

3 Field choice

Having settled the curve shape and generation method, the remaining major choice is the field. Following Safecurves, I used a prime-order field. That leaves at least six families of desirable primes:

[1]Similarly, for ciphers with $>$ 128-bit keys, the security margin is more informative than the actual key length.

3.1 Possible prime shapes

Random primes. Brainpool-style random primes are an interesting option, as they would preclude hypothetical attacks based on special field forms. However, this comes at a severe performance impact, and it is more difficult to argue that the coefficients have been chosen randomly rather than maliciously. What's more, lacking any information about attacks on "fast" fields, it seems better to spend performance on a larger field instead of on a random field. So I chose a large, fast field instead.

Mersenne primes. The Mersenne prime $2^{521} - 1$ is a good option, with very good performance for its size. I implemented and tested the curve E-521 on Haswell, with benchmarks shown in Section 6. But this prime is enormous, and I believe that a less extreme level of overkill would allow wider deployment.

Crandall primes ($2^k - c$). These are a very popular option, and for good reason. However, they have an important constraint to get the best possible performance. To limit carry propagation during multiply-reduce, c should be small. This limits the fastest Crandall choices to a few options, such as $2^{379} - 19$, $2^{389} - 17$, and $2^{414} - 17$.

Special Montgomery primes ($2^k c - 1$). The advantage of special Montgomery primes is mainly in carry propagation on full-radix implementations. But primes greater than about 2^{256} favor vectorized multiplication on ARM and reduced-radix multiplication on x86-64, and these require changes to carry propagation anyway. So this prime shape loses its advantage, and has several disadvantages, particularly more complex direct reduction and more bias from hash-truncate-reduce.

Granger-Moss primes ($\Phi_N(k)$). These primes are very fast on 64-bit machines at around 240 or 360 bits [13], and they vectorize reasonably well. But they require one specific implementation method, which means that they may suffer performance problems on platforms other than their target platform. Furthermore, these primes do not work as well past 360 bits of field size. This is essentially because 9 is not prime, so that Φ_9 has degree only 6.

Solinas primes ($2^k - 2^\ell \pm \ldots \pm 1$). These primes can be very fast, but only if they have few coefficients and those coefficients are on powers of the radix. For example, the NIST primes have 32-bit-aligned exponents, so they work better on 32-bit platforms using a 32-bit radix. Because I wanted a curve which would be fast on multiple platforms, I selected a prime with as few coefficients as possible (3, for non-Mersenne primes) and which aligns to several different radices.

3.2 The Goldilocks prime, $2^{448} - 2^{224} - 1$

I chose the Solinas trinomial prime $p := 2^{448} - 2^{224} - 1$. I call this the "Goldilocks" prime because its form defines the golden ratio $\phi \equiv 2^{224}$. Because $224 = 32 \cdot 7 = 28 \cdot 8 = 56 \cdot 4$, this prime supports fast arithmetic in radix 2^{28} or 2^{32} (on 32-bit machines) or 2^{56} (on 64-bit machines). With 16, 28-bit limbs it works well on vector units such as NEON. Furthermore, radix-2^{64} implementations are possible with greater efficiency than most of the NIST primes.

Karatsuba The main advantage of a golden-ratio prime is fast Karatsuba multiplication. Let $\phi = 2^{224}$ as above. Then

$$\begin{aligned}
&(a + b\phi) \cdot (c + d\phi) \\
&\quad = ac + (ad + bc)\phi + bd\phi^2 \\
&\quad \equiv (ac + bd) + (ad + bc + bd)\phi \pmod p \\
&\quad = (ac + bd) + ((a + b)(c + d) - ac)\phi
\end{aligned}$$

This can be evaluated particularly efficiently by considering individual limbs spaced apart by ϕ at the same time.

Implementation on 32-bit platforms The coefficients of this prime are 32-bit aligned, so it should admit a fast packed radix-2^{32} implementation with 14 limbs. But even on most 32-bit platforms, this may not be the best option. Carry propagation is expensive, especially with a vector unit. Instead, the prime can be implemented with radix 2^{28} and 16 limbs. If each limb enters the multiplication routine less than $2^{28} \cdot c$ for some c, then the largest coefficient before reduction will be at most $38 \cdot 2^{56} \cdot c^2$. This will be less than 2^{64} if $c < 2^4/\sqrt{38} > 5/2$. A tiny correction is required to prevent overflows during carry propagation itself, but the result is still $> 5/2$. So implementers can begin with coefficients reduced to less than $2^{28} \cdot 5/4$, and can still perform one unreduced addition to each multiplicand without the possibility of overflow.

Karatsuba multiplication does not change this analysis so long as the strategy is to compute each coefficient and then reduce. Even if the accumulator overflows and underflows during the computation, it will still end up correct. Overflow can still cause problems with saturating arithmetic, such as ARM NEON's VQDMLAL, so in practice c is bounded more tightly around 2.

The resulting formulas vectorize well, at least on ARM NEON. A signed-limb representation might reduce the cost of subtraction at some cost during reduction, but I haven't tested this.

Implementation on 64-bit platforms Likewise, 64-bit implementations can use radix 2^{56} and 8 limbs. But they can add and subtract many times without the possibility of overflow, magnifying reduced limbs by a factor of up to $41 < 2^8/\sqrt{38}$. Vectorization is less important on 64-bit platforms because the integer scalar multiplier is usually the fastest one on the core. But vectors are still useful for the initial Karatsuba additions, for the add/sub routines and for constant-time selection and swapping.

If only 64-bit machines need to be supported, then the even larger "Ridinghood" prime, $2^{480} - 2^{240} - 1$, gives almost as good performance as Goldilocks.

Elligator compatibility and nonce choice Note that the bias in a random variable created by simply choosing a 448-bit input and reducing mod p is about 2^{-224}, or $1/\sqrt{p}$. So noticing this bias is comparably expensive to a rho-attack on the curve. This means that p is easy to use with Elligator [5].

Likewise, the order $4 \cdot q$ of any curve modulo p is within $3 \cdot 2^{224}$ of 2^{448} by Hasse's bound. Therefore it is safe to produce scalars by truncating a random string to 446 bits. This bias mod q from this method is a negligible $3 \cdot 2^{-224}$. The more conservative method of choosing a random 512-bit number and reducing modulo q is also safe.

4 Curve coefficients

I chose an untwisted Edwards curve, i.e. one of the form

$$E_d : y^2 + x^2 = 1 + dx^2y^2$$

To demonstrate that there is nothing up my sleeve, I chose d as small as possible in absolute value so that E_d and its twist both have 4·prime order, and so that the order of the curve is less than p. This last restriction was for ease of implementation, but it doesn't matter because the least d already gives a curve of order less than p.

The Goldilocks d is -39081. The resulting curve satisfies the other Safecurves criteria, as shown on `safecurves.cr.yp.to`. In particular, because d is not square in $\mathbb{Z}/p\mathbb{Z}$, the strongly unified Edwards point addition formulas apply.

The order of the curve is

$$4 \cdot q := 4 \cdot \left(2^{446} - \left(\begin{array}{r} \texttt{0x8335dc163bb124b651} \setminus \\ \texttt{29c96fde933d8d723a7} \setminus \\ \texttt{0aadc873d6d54a7bb0d} \end{array}\right)\right)$$

and the order of its twist is

$$4 \cdot \left(2^{446} + \left(\begin{array}{l} \texttt{0x335dc163bb124b651} \setminus \\ \texttt{29c96fde933d8d723a7} \setminus \\ \texttt{0aadc873d6d54a7bb0d} \end{array}\right)\right)$$

CFRG has specified the generator on the 4-isogenous Montgomery curve to be the order-q point with the least u-coordinate ($u = 5$), as in Curve25519. This corresponds to a generator of the Edwards curve, namely

$$\left(\frac{\sqrt{5}}{3}, \begin{array}{r} \texttt{0x51fa169cb528fb724ca6} \setminus \\ \texttt{29dfaf793d4ffc91285fca7} \setminus \\ \texttt{7b228481c928c75273b47f2} \setminus \\ \texttt{9a9a7cc5d5cf6744434d412} \setminus \\ \texttt{e325f9425150432156c7912} \end{array}\right)$$

where due to the golden ratio prime shape,

$$\sqrt{5}/3 = \begin{array}{r} \texttt{0x5555555555555555555555555555} \setminus \\ \texttt{5555555555555555555555555555} \setminus \\ \texttt{aaaaaaaaaaaaaaaaaaaaaaaaaaaa} \setminus \\ \texttt{aaaaaaaaaaaaaaaaaaaaaaaaaaaa} \end{array}$$

5 Implementation

I have written and published an implementation of Ed448-Goldilocks. The implementation on most platforms uses C with assembly intrinsics for multiply-and-accumulate, and compiler extensions for vector arithmetic. However, the NEON implementation uses an assembly implementation of a complete field multiplication. This is because on most platforms, I was able to achieve nearly assembly-level performance with a mostly-C-level implementation, but on NEON I was unable to do so.

The implementation uses the "Decaf" strategy [16] to implement a prime-order group $2\mathcal{E}/\mathcal{E}[2]$ from the cofactor-4 curve. It includes a library with general point operations, complete point and scalar arithmetic, scalar multiplication, point compression, and invertible maps to the curve. It includes several example crypto primitives.

The Decaf strategy works the same on an Edwards curve and on the 4-isogenous Montgomery and twisted Edwards curves. Therefore the implementation internally uses the slightly faster twisted Edwards form, with $a = -1$ and $d = -39082$.

Key generation The key generation algorithm uses the signed all-bits-set combs algorithm. On 64-bit platforms it uses 15kiB of tables (5 comb tables, 5 teeth per comb, 3 coordinates per point, expanded to 32 bytes per coordinate). On 32-bit platforms it instead uses 12kiB of tables (8 combs, 4 teeth per comb).

Signing The signature algorithm produces Schnorr signatures. It uses the same combset as the key generation algorithm.

Verification The verification algorithm runs in variable time, using wNAF and a 6kiB precomputed table with multiples of the generator. Verification can in principle be batched, but I haven't tested this.

ECDH ECDH uses the Montgomery ladder, which is marginally faster than an Edwards scalar multiplication because it doesn't need to decompress the input point. The Montgomery ladder is enhanced to preserve sign information and to reject points on the twist, even though it is believed to be safe to allow them. It is therefore a drop-in replacement for an Edwards scalar multiplication.

5.1 Source code

The source code for my Ed448-Goldilocks implementation is available at `http://sourceforge.net/p/ed448goldilocks/code/ci/decaf/tree/` under an MIT license.

6 Benchmarks

To assess the performance of Ed448-Goldilocks vs. alternative curves, I ran key generation, ECDH, signing and verification tests on each alternative on several benchmarking machines.

I measured seven different curves. It is important to note that the implementation effort, time/memory tradeoffs, particular functions measured and measurement technique differ by curve and by platform.

- Goldilocks. This is a prime-order sub/quotient group of the Ed448-Goldilocks curve, using the Decaf point compression method to remove the cofactor. It is implemented in C on x86-64 and ARM scalar, with intrinsics for word multiply and accumulate. On ARM NEON, the entire field multiplication and squaring routines are written in assembly, because my attempts with intrinsics gave poor performance. I measured performance with supercop-fastbuild [6].

- E-521. This is a $2q$-order subgroup of E-521, from on an older branch of the Ed448-Goldilocks code base. It implements point compression, but not complete cofactor removal. It is implemented only on x86-64 using C with intrinsics, and is specifically tuned for processors with AVX2. I measured performance with `make bench`.

	Haswell	A9	A8+NEON
`goldilocks_shared_secret`	531k	3.5M	1.8M
`goldilocks_keygen`	166k	1.5M	599k
`goldilocks_sign`	162k	1.5M	632k
`goldilocks_verify`	588k	3.8M	1.9M
`e521_shared_secret`	803k		
`e521_keygen`	234k		
`e521_sign`	243k		
`e521_verify`	875k		
`ed-384-mers-variable (ecdh)`	500k	9.9M	7.9M
`ed-384-mers-fixed  (keygen)`	185k	3.2M	2.7M
`ed-384-mers-db     (verify)`	508k	9.9M	7.6M
`ed-512-mers-variable (ecdh)`	1.1M	22 M	20 M
`ed-512-mers-fixed  (keygen)`	357k	6.5M	6.5M
`ed-512-mers-db     (verify)`	1.1M	22 M	21 M
`openssl ecdh-p256`	236k	4.4M	4.8M
`openssl ecdsa-sign-p256`	133k	1.2M	1.3M
`openssl ecdsa-verify-p256`	333k	5.2M	5.8M
`openssl ecdh-p384`	2.1M	11.1M	12.8M
`openssl ecdsa-sign-p384`	618k	2.7M	3.0M
`openssl ecdsa-verify-p384`	2.6M	13.2M	15.4M
`openssl ecdh-p521`	1.8M	24.2M	29.3M
`openssl ecdsa-sign-p521`	1.1M	5.4M	6.3M
`openssl ecdsa-verify-p521`	2.5M	29.7M	34.4M

Figure 1: Ed448-Goldilocks benchmarks vs OpenSSL 1.0.1f / 1.0.2 and MSR ECCLib 1.2. The Haswell measurements are on a Core i7-4790 at 3.6GHz with hyperthreading and TurboBoost disabled, and OpenSSL 1.0.2. The Cortex A9 numbers are from a 1GHz Tegra 2 processor in a TrimSlice, with OpenSSL 1.0.1f. The Cortex A8+NEON numbers are on an AM335x BeagleBone Black at 1GHz with OpenSSL 1.0.1f.

The E-521 code is particularly sensitive to compiler options, possibly due to alignment sensitivity. I've reported the most favorable time for ECDH, 803kcy, but different versions and compilation flags give up to 860kcy on the same machine.

- ed-384-mers. This is the ed-384-mers group with cofactor 4, as implemented in MSR ECCLib 1.2. On x86-64, all field arithmetic is implemented entirely in assembly language, but without the (small) benefit of the Haswell BMI2 intrinsics. On ARM it is implemented in C. MSR ECCLib does not implement point compression, which gives it an advantage in signature verification speed. However, its key generation algorithm is not as optimized as the one Goldilocks uses, and uses smaller precomputed tables. I measured performance with `./ecc_tests`.

  On ARM, I did minimal modification of the compilation flags and source to ensure that -mcpu was set correctly and that the multiplication and squaring inner loops were unrolled, but the code should not be regarded as optimized.

- NIST secp256r1, secp384r1, secp521r1. These are implementations of short Weierstrass curves without point compression. They include the Gueron and Krasnov [14] optimizations for secp256r1 on x86-64, but the ARM implementations are relatively unoptimized. I measured performance with `openssl speed`.

The benchmarks are shown in Figure 1. The benchmarks are taken with hyperthreading and TurboBoost disabled. The software used in this benchmark varies widely, so it is important to take these numbers with a grain of salt. It is seen that both Ed448-Goldilocks and MSR ed-384-mers are around 4x as fast as OpenSSL's implementation of NIST-P384 on x86-64, and on ARM Goldilocks is faster even than OpenSSL's NIST-P256. Additionally, Ed448-Goldilocks can take advantage of ARM NEON for a $\approx 2\times$ speedup.

Compared to ed-384-mers, Ed448-Goldilocks is some 17% faster for key generation, 16% slower for verification and 6% slower for ECDH. These differences are exaggerated by implementation details. Goldilocks' use of point compression slows down verification, and its larger and better-tuned combset make it faster on key generation.

Unfortunately, while Ed448-Goldilocks had 32-bit performance as a design requirement, no fair comparison is available on 32-bit platforms because the implementations are too different. One could compare reference implementations, where Goldilocks and ed-384-mers take about the same amount of time: some 11-13 million Cortex-A9 cycles for ECDH depending on the source branch and compiler. But this is mostly a benchmark of which optimizations are still present in the reference implementation.

When designing Goldilocks, I hypothesized that timings within the same implementation strategy should roughly follow a power law $t \propto b^k$, where b is the bits in the field size and k is some exponent, and therefore that b^k/t might be described as an "efficiency score". I evaluated designs with Karatsuba's $k = 1 + \log 3/\log 2$, which works surprisingly well even though not all the designs use Karatsuba multiplication. In arbitrary units[2], Ed448-Goldilocks and E-521 score 3.3 and 3.1 by this metric. ECCLib's ed-384-mers and ed-512-mers both score 2.3, and the optimized secp256r1 scores 1.4. By this metric, Goldilocks is about 40% more efficient than either ed-mers curve on Haswell.

7 Conclusions

Ed448-Goldilocks is a conservatively designed elliptic curve with very competitive performance on a wide variety of platforms. It is a suitable choice for standardization as an alternative to both secp384r1 and secp521r1.

References

[1] Diego F. Aranha, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira, and Jefferson E. Ricardini. A note on high-security general-purpose elliptic curves. Cryptology ePrint Archive, Report 2013/647, 2013. `http://eprint.iacr.org/`.

[2] Daniel Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. *Progress in Cryptology–AFRICACRYPT 2008*, pages 389–405, 2008.

[3] Daniel Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In *Advances in cryptology–ASIACRYPT 2007*, pages 29–50. Springer, 2007.

[2]The units are b^k/t with t the total number of Haswell cycles for a one-side-signed DH handshake: two keygens, two ECDH's, a signature and a verification.

[4] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 316–334. Springer Berlin Heidelberg, 2014.

[5] Daniel J Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: Elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 967–980. ACM, 2013.

[6] Daniel J. Bernstein, Tanja Lange, and John Schanck. supercop-fastbuild, 2014. `https://github.com/jschanck-si/supercop-fastbuild`.

[7] D.J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.Y. Yang. High-speed high-security signatures. *Cryptographic Hardware and Embedded Systems, CHES 2011*, 2011.

[8] D.J. Bernstein and P. Schwabe. Neon crypto, March 20 2012. `http://cryptojedi.org/papers/neoncrypto-20120320.pdf`.

[9] Joppe W. Bos, Craig Costello, Patrick Longa, and Michael Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. Cryptology ePrint Archive, Report 2014/130, 2014. `http://eprint.iacr.org/`.

[10] ECC Brainpool. Ecc brainpool standard curves and curve generation, 2005. `http://www.ecc-brainpool.org/download/Domain-parameters.pdf`.

[11] H.M. Edwards. A normal form for elliptic curves. *Bulletin-American Mathematical Society*, 44(3):393, 2007.

[12] S. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Advances in Cryptology-EUROCRYPT 2009*, pages 518–535, 2009.

[13] Robert Granger and Andrew Moss. Generalised mersenne numbers revisited. Cryptology ePrint Archive, Report 2011/444, 2011. `http://eprint.iacr.org/`.

[14] Shay Gueron and Vlad Krasnov. Fast prime field elliptic curve cryptography with 256 bit primes. Cryptology ePrint Archive, Report 2013/816, 2013. `http://eprint.iacr.org/`.

[15] Mike Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, 2012. `http://eprint.iacr.org/2012/309`.

[16] Mike Hamburg. Decaf: Eliminating cofactors through point compression. *To appear in CRYPTO 2015*, 2015.

[17] Hüseyin Hışıl. Elliptic curves, group law, and efficient computation, 2010.

[18] Hüsseyin Hışıl, Kenneth Wong, Gary Carter, and Ed Dawson. Twisted edwards curves revisited. *Advances in Cryptology–ASIACRYPT 2008*, pages 326–343, 2008.

[19] G Locke and P Gallagher. Fips pub 186-3: Digital signature standard (dss). *Federal Information Processing Standards Publication*, 2009.

[20] Alexey Melnikov. [cfrg] results of the poll: Elliptic curves - preferred curves around 256bit work factor (ends on march 3rd). `http://www.ietf.org/mail-archive/web/cfrg/current/msg06398.html`.

[21] Michael Scott. A new curve. https://moderncrypto.org/mail-archive/curves/2015/000449.html.

[22] Christopher A. Taylor. Snowshoe: Portable, secure, fast elliptic curve math library in c. https://github.com/catid/snowshoe.