Truncated EdDSA/ECDSA Signatures

Thomas Pornin

NCC Group, thomas.pornin@nccgroup.com

19 July, 2022

Abstract. This note presents some techniques to slightly reduce the size of EdDSA and ECDSA signatures without lowering their security or breaking compatibility with existing signers, at the cost of an increase in signature verification time; verifying a 64-byte Ed25519 signature truncated to 60 bytes has an average cost of 4.1 million cycles on 64-bit x86 (i.e. about 35 times the cost of verifying a normal, untruncated signature).

1 Signature Size Reduction

We consider the following situation:

- A system involves a *signer*, who generates digital signatures on some data. The signatures must then be transmitted to a *verifier*, who validates the signatures against the signer's public key and the purported signed data.
- There are severe constraints on the size of the signature, due to the transmission mechanism. For instance, the signature and some other data must fit in a QR code. Another possible case would be the packaging of many signed transactions in a limited-bandwidth ledger, as is common in blockchain systems.
- At the same time, the signature generation algorithm must be one of a few standard mechanisms, for compatiblity with the signer's hardware (e.g. it is a smartcard) or for compliance reasons.

In such a situation, one would want a signature algorithm that minimizes signature size. A standard Ed25519 signature (EdDSA over Curve25519) or ECDSA (with NIST's curve P-256) has size 64 bytes. If the algorithm can be chosen freely, then various possibilities to reduce the signature size exist, e.g.:

- Use a smaller elliptic curve; for instance, NIST's P-192 still offers a formidable and currently unbreakable security level of 96 bits, and yields ECDSA signatures of 48 bytes.
- Employ an EC-based Schnorr signature with a reduced "challenge" size. An n-bit security level can be obtained with a $2n$-bit curve and an n-bit hash function output for the computation of the internal challenge, as long as the hash function is not a "narrow-pipe" design[6]. This would allow Schnorr signatures with a 256-bit curve and a purported 128-bit security level to fit in 48 bytes or so.
- Use a pairing-friendly curve and the BLS signature scheme[2]. A suitable curve might provide a base curve with a prime order of about 2^{256} elements, and yield signatures of size 32 bytes only with a 128-bit security level. For various reasons (notably performance), somewhat larger curves are currently being standardized[9], but a signature size of 48 bytes is obtained with curve BLS12-381.

However, if we are stuck with standard Ed25519 or ECDSA over P-256, then such solutions cannot be applied. In this note, we explore another method, which is to simply truncate the signature, and let the verifier rebuild the missing part during the verification process.

Generic Truncation. In general, some truncation can be safely applied on *any* signature scheme: the last t bits of the signature are omitted, and the verifier just tries all possible 2^t values of the missing bits until a valid signature is obtained. This entails running the verification algorithm an average of 2^{t-1} times. The process is safe for the following reasons:

- The truncation and reconstruction use only public data.
- A valid untruncated signature is obtained as a byproduct.

In other words, if a forgery attack is feasible against the truncated signature scheme, then the attacker can simply run this validation process on that forged truncated signature to obtain an untruncated forgery as well. In that sense, support for truncated signatures cannot make forgery attacks easier.

Of course, the verification cost rises quite steeply with the number of removed bits, so that a practical implementation would probably not be able to remove more than 8 or 10 bits before the verification becomes too expensive. However, in the case of EdDSA and ECDSA signatures, some optimizations are feasible, allowing practical truncation of, for instance, 32 bits, reducing the signature size from 64 down to 60 bytes. The purpose of this note is to describe these optimizations, in the case of both Ed25519, and ECDSA with NIST's P-256. We implemented and benchmarked both cases; the implementation is available in the open-source `crrl` library, available on:

`https://github.com/pornin/crrl/`

2 EdDSA Signatures

Notations. EdDSA is standardized in RFC 8032[5] for twisted Edwards curves; we focus here on Ed25519. In the description below, we use the following notations:

- The public key is the curve point A.
- The conventional generator for the curve subgroup is B.
- The signature consists of (the encoding of) a point R and a scalar s. The scalar is an integer modulo the curve subgroup order $\ell \approx 2^{252}$. R and s are encoded in that order, over 32 bytes each; s uses little-endian.
- The verification algorithm entails computing a challenge value k, which is a scalar; it is obtained by using a hash function over the concatenation of the point R, the public key A, and the signed message. The scalar k may have a value anywhere between 0 and $\ell - 1$. The signature is then deemed valid if the following equation is fulfilled:

  $$8sB = 8R + 8kA$$

  Note that we use here the "cofactored" equation (with the multiplier 8). A "cofactorless" equation is also defined, without that multiplier, but its use is not recommended (notably, some optimized verification mechanisms, in particular batch verification, cannot easily enforce the validity of the cofactorless equation).

Core Verification Algorithm. Since s uses the little-endian encoding convention, removing the last t bits of the signature is equivalent to reducing the integer s (nominally in the 0 to $\ell - 1$ range) modulo 2^n, for $n = 256 - t$. We suppose here that $t \geq 8$; in practice, our implementation supports any t from 8 to 32. Note that the top three bits of s are already known to be zero, since $\ell < 2^{253}$.

Let s_0 be the value as received, i.e. $s_0 = s \bmod 2^n$. We can write:

$$s = s_0 + 2^{251} + s_1 2^n$$

for an integer s_1 such that $-2^{t-5} \leq s_1 \leq +2^{t-5}$ (the value $+2^{t-5}$ is possible in case $s \geq 2^{252}$, because ℓ is slightly *above* 2^{252}). The verification equation can then be rewritten as:

$$s_1(2^n 8B) = 8R + 8kA - 8(s_0 + 2^{251})B$$

By defining $U = 2^{n+3}B$ and $V = 8(R+kA-(s_0+2^{251})B)$, the truncated signature verification becomes the problem of finding a solution s_1 in the $[-2^{t-5} \ldots +2^{t-5}]$ range to the restricted discrete logarithm problem:

$$s_1 U = V$$

In general, we can hope for solving such a problem with a cost proportional to the square root of the width of the range of possible solutions. Here, we can furthermore leverage the following facts:

- In a twisted Edwards curve such as Curve25519, points have coordinates x and y; for any point P, points P and $-P$ have the same y coordinate.
- The point U defined above depends on the number of removed bits (t) but not on the signature value nor the public key; this allows part of the computation to be performed in advance, and included in the verifier software as a precomputed table.

Here are the steps that we use in our implementation:

1. Let I and J be two positive integers such that $IJ \geq 2^{t-5}$. Exact values will be discussed below; in general, one can think of I and J to be roughly equal to $2^{(t-5)/2}$.
2. For $j = 0$ to J, define $U_j = jIU$. We compute and accumulate the y coordinates of points U_j in a table, that we sort so that lookups are efficient.
3. For $i = 0$ to $I - 1$, define $V_i = V - iU$. We compute the points V_i and extract their y coordinates.
4. We look for a match between the y coordinates of a point U_j and a point V_i. For any match, we have two candidates for the solution: $s_1 = i + Ij$ and $s_1 = i - Ij$. Each candidate is validated against the normal (untruncated) verification equation.

If a solution s_1 exists (i.e. the signature is valid), then there must exist two integers a and b such that $s_1 = a + Ib$, with $0 \leq a < I$ and $-J \leq b \leq J$. In that case:

$$(a + Ib)U = V$$
$$bIU = V - aU$$

Since $U_{-b} = -U_b$, points U_b and U_{-b} have the same y coordinate. Therefore, our search process must find a match for $i = a$ and $j = |b|$, and we will try $s_1 = a + Ib$ as a potential solution.

Precomputations. Since U does not depend on runtime parameters, we can precompute the values U_j and store an already sorted search table in the code.

The validation of each possible solution s_1 is relatively fast since s_1 is a small integer, and U and V do not have to be recomputed each time; it is much faster than a complete verification. We can therefore afford a few false positives. Correspondingly, we can store only a few bits of the y coordinates of the U_j in the precomputed table (in our implementation, we keep 48 bits only of each y coordinate). Thus, for a given code footprint budget, we can use a somewhat larger table.

The runtime cost is roughly proportional to I (for the computation of the points V_i); the cost induced by a large J is only static read-only storage (for the precomputed table) and $O(\log J)$ for each lookup (with a binary search). Thus, it is advantageous to make J larger than I. In our implementation, we use $J = 2^{14}$, i.e. a table of 16385 elements (about 131 kB), and I can range up to $I = 2^{13}$ (for the maximum supported truncation of $t = 32$ bits).

Using the Montgomery Curve. General point addition on a twisted Edwards curve uses eight multiplications in the base field (8M). However, we can lower the per-V_i cost by switching to the Montgomery domain, i.e. mapping the points to the birationally equivalent Montgomery curve, where we can have a per-point cost of 6M instead. Specifically, the twisted Edwards point $P = (x, y)$ is mapped to a Montgomery point (e, f) with $e = (1 + y)/(1 - y)$. Then, if the e coordinates of points U, V_i and V_{i+1} are the fractions N_u/D_u, N_i/D_i and N_{i+1}/D_{i+1} respectively, then we get the u coordinate of V_{i+2} as a fraction N_{i+2}/D_{i+2} with the following equations:

$$N_{i+2} = D_i((N_{i+1} - D_{i+1})(N_u + D_u) + (N_{i+1} + D_{i+1})(N_u - D_u))^2$$
$$D_{i+2} = N_i((N_{i+1} - D_{i+1})(N_u + D_u) - (N_{i+1} + D_{i+1})(N_u - D_u))^2$$

For more details on these formulas, see [4] (section 3.2).

Batch Inversion. Since we get the point coordinates as fractions, they must be normalized to affine values, which entails a division in the base field. This is an expensive operation; however, several inversions can be batched together by applying recursively a trick due to Montgomery ($1/a = b(1/(ab))$ and $1/b = a(1/(ab))$). This method allows inverting n field elements for the cost of a single inversion in the field, and an additional $3(n - 1)$ multiplications. Since inversion cost can range up to 100 or more multiplications in the field, the batch method is a huge gain. In our implementation we use batches of 200 values.

Early Exit. We do not have to compute all V_i before starting to lookup coordinates in the table of U_j values; we can do so as the V_i are obtained (subject to the granularity of batch inversion for normalization to affine coordinates). Since signature verification uses only public data, there can be no secret information leak and we can exit as soon as we obtain a valid signature. Thus, on average, we will need to compute only half of the V_i before finding the right solution; the worst case (when the match is found only last, or when there is no match at all because the signature is invalid), the cost will be up to twice that of the average.

Extra Bits in the Commitment. If we are desperate for size, then up to 3 extra bits of information can be smuggled into the commitment part of the signature, i.e. the point R. In a

normal, legitimately produced signature, the point R is generated by using a (pseudo)random scalar r, i.e. an integer in the 0 to $\ell - 1$ range, and computing $R = rB$. The point B has order ℓ. However, the complete Curve25519 has order 8ℓ, and B generates only a subgroup of the curve. In general, any point P on the curve can be written uniquely as $P = P_\ell + P_8$, with P_ℓ being part of the subgroup of order ℓ, and P_8 being an element of a subgroup of order 8 (denoted $\mathcal{E}[8]$). We therefore expect $R = R_\ell + R_8$ to be such that $R_8 = (0, 1)$ (the neutral element on the curve).

This leads us to the following scheme:

- Let T be a generator of $\mathcal{E}[8]$. There are four such points; any will do.
- An extra 3-bit information g (as an integer such that $0 \leq g < 8$) is encoded into the signature by replacing R with $R' = R + gT$. Since we use the cofactored verification equation, this does not make the signature invalid.
- Information g can be recovered by multiplying the received R' by ℓ:

$$\ell R' = \ell R + \ell g T = (\ell g \bmod 8)\, T$$

It shall be noted that this scheme relies on the assumption that the original R is indeed in the subgroup of order ℓ. It could be argued that a signer could legitimately issue valid signatures with R not in that subgroup (and correspondingly generate public/private key pairs with the public key not in the subgroup either) as long as they still validate with the signature verification equation. There does not seem to be any good reason to do so, though. To cover that case, the information embedding step may first normalize R to the proper subgroup (by multiplying it by $8^{-1} \bmod \ell$, then by 8), but this implies some extra costs.

We did *not* implement this method because it involves curve point computations on the signer's side. In situations where a hardware signature generator must be used (e.g. a smartcard), there might not be easily available implementations of such operations outside of the signing engine; the signer might furthermore lack the computing resources to do so.

Performance. We implemented the truncated signature support in `crrl`, which is written entirely in the Rust language. No inline assembly is used, nor SIMD opcodes (e.g. AVX2 or NEON); on x86 systems, the `_addcarry_u64()` and `_subborrow_u64()` intrinsics are used, since they seem to provide a slight speed-up over the portable constructions that we use on other architectures. All operations on secret data are fully constant-time. Curve operations use regular extended coordinates, with some optimizations for sequences of doublings. Untruncated signature verification uses the optimization described by Antipa *et al* in [1], with the optimized lattice reduction algorithm from [7].

We benchmarked the implementation for various truncations (removal of 8, 16, 24, 28 or 32 bits). Rust compiler 1.59 is used ("stable" channel). Benchmark uses the "release" optimization level, with the extra flags "`-C target-cpu=native`" to specifically target the system on which the process is executed. The two test systems are:

- An Intel i5-8259U running at 2.3 GHz, in 64-bit mode (x86-64 architecture), under Linux (Ubuntu 22.04). TurboBoost is disabled.
- A Raspberry Pi, model 3B, with a BCM2837 CPU (ARM Cortex A53 core), running at 1.2 GHz, in 64-bit mode (aarch64 architecture), under Linux (Ubuntu 20.04.4). Performance counters were enabled and used to get accurate cycle counts.

Results are summarized in table 1. We note that even with the maximal truncation (32 bits), the signature verification cost is only 25 to 35 times (on average) that of the normal, untruncated signature verification (for valid signatures).

Operation	**x86-64 (Intel "Coffe Lake")**		**aarch64 (ARM Cortex A53)**	
	valid signature	invalid signature	valid signature	invalid signature
Sign	51497	-	212906	-
Verify	114031	114031	478875	478875
Verify trunc ($t = 8$)	150734	149270	671503	662950
Verify trunc ($t = 16$)	152607	146079	680137	651421
Verify trunc ($t = 24$)	177044	170473	761440	711991
Verify trunc ($t = 28$)	417015	585547	1541250	1956374
Verify trunc ($t = 32$)	4086200	7399430	12008249	21844606

Table 1: Performance of Ed25519 truncated signatures. Values are in clock cycles. Signature verification measures are averages for either valid signatures, or invalid signatures.

3 ECDSA Signatures

The process for truncated ECDSA signatures is similar to that of EdDSA, but with some differences that we discuss below. We focus on the standard NIST curve P-256 (also known as "secp256r1" or "prime256v1").

Notations The curve has order ℓ, which is close to (but lower than) 2^{256}. It is a prime integer; there is no cofactor. The curve conventional generator point is denoted G. The public key is the point Q. A signature is a pair (r, s) of integers modulo ℓ.

In a normal ECDSA signature verification, the input message is hashed, and the hash value is converted (through truncation and modular reduction) into a scalar m. The verification then consists of computing the point $R = (m/s)G + (r/s)Q$, and finally verifying that the x coordinate of the point R is equal to r (modulo ℓ). Note that the x coordinate of point R is in the base field in which the curve is defined, i.e. it is an integer modulo a given prime p which is close to, but distinct from, ℓ. The transform of that value into r thus requires first turning it into an integer representative (in the 0 to $p - 1$ range), then reducing that integer modulo ℓ.

ECDSA signatures can be encoded in several ways; one of the most common formats uses unsigned big-endian encoding of r and s, concatenated in that order. The two encodings are left-padded if necessary to ensure that they have the same size (in bytes), so that their interpretation by the verifier is unambiguous. With curve P-256, this yields signatures of 64 bytes (about $1/2^{16}$ of signatures could be encoded in 62 bytes or fewer, because the top bytes of r and s would happen to be both zero, but in practice we must assume that signatures have size 64 bytes).

Preparation for Truncation. In order to help with truncation, a few inexpensive steps are first applied to the signature (r, s):

- If $r < p - \ell$, then the signature is rejected. This step ensures that there is a unique value for the x coordinate of point R that matches r (equivalently, that the reduction of the integer modulo ℓ did not alter the value). The probability that a legitimately generated ECDSA signature has a value r in the rejected range is about $2^{-128.9}$, which means that it does not really happen in practice, and if it ever happens, then it is almost surely due to a hardware failure rather than a singular stroke of bad luck.
- If $s \geq 2^{255}$ then it is replaced with $\ell - s$. This leverages the fact that if (r, s) is a valid signature, then so is $(r, -s)$, and vice versa; this is due to the property of short Weierstraß curves that points R and $-R$ have the same x coordinate. This allows us to ensure that s can be encoded over 255 bits.
- s is reencoded using the little-endian convention, so that truncation applies to the most significant bits rather than the least significant.

After these steps, the signature value can be truncated, by removing the last t bits (for some value of t, typically at most 32 bits).

Core Verification Algorithm. Due to the restriction we applied on the value of r, the x coordinate of point R can be rebuilt unambiguously. This allows recomputing the point R (through a process analogous to standard point decompression), although we lack its sign; thus, wo do not know if the point R we recompute is the real R, or $-R$. This does not matter in the rest of the process, though it prevents us from applying an optimization that we could use in the case of EdDSA.

Write $s = s_0 + s_1 2^n$, for $n = 256 - t$. Since the preparation step ensured that $s < 2^{255}$, we know that $0 \leq s_1 < 2^{t-1}$. The verification equation can be rewritten as:

$$sR = mG + rQ$$

which leads to:

$$s_0 R + s_1(2^n R) = mG + rQ$$

As in the EdDSA case, we define two sets of points U_j and V_i and look for a match, i.e. two points that have the same x coordinate. In more details:

1. Let I and J be positive integers such that $IJ \geq 2^{t-1}$. As will be explained below, we normally choose $J = 2^{(t-2)/2}$ and $I = 2J$.
2. For $j = 0$ to J, define $U_j = s_0 R + jI(2^n R)$. We compute and accumulate the x coordinates of the U_j in a table sorted for fast lookups.
3. For $i = 0$ to I, compute $V_i = mG + rQ - i(2^n R)$ and verify whether the x coordinate of V_i is equal to the x coordinate of one of the U_j. If a match is found between U_j and V_i, then the two candidates are $s = s_0 + (i + Ij)2^n$ and $s = s_0 + (-i + Ij)2^n$.

If a solution s_1 exists, then we can write $s_1 = a + Ib$ with $0 \leq a < I$ and $0 \leq b < J$. If we rebuilt the correct R, then the verification equation is:

$$s_0 R + (a + Ib)(2^n R) = mG + rQ$$

which leads to $U_b = V_a$, thus a match with $i = a$ and $j = b$, and $s = s_0 + (i + Ij)2^n$. On the other hand, if we rebuilt $-R$ instead of the correct R, then the verification equation becomes:

$$-(s_0R + (a + Ib)(2^nR)) = mG + rQ$$

which can be rewritten as:

$$-(s_0R + I(b + 1)(2^nR)) = mG + rQ - (I - a)(2^nR)$$

i.e. $-U_{b+1} = V_{I-a}$, so that we will get a match between the x coordinates of U_{b+1} and V_{I-a}. We then have $i = I-a$ and $j = b+1$, and the solution is $s = s_0+(I-i+I(j-1))2^n = s_0+(-i+Ij)2^n$. Note that we included the upper range limit on i and j (i.e. we computed U_J and U_I as well) precisely so that $I - a$ and $b + 1$ are always part of the covered indices.

The verification process can terminate as soon as the solution is found; for a valid signature, about half of the V_i (on average) have to be computed, but all the U_j, which is why it is advantageous to choose $I = 2J$ (as an approximation, the total cost is proportional to $J + I/2$, which is minimized for $I = 2J$ when the product IJ is fixed).

X-only Arithmetics. As in the EdDSA case, we only need the x coordinates of the U_j and V_i points, and we can obtain them with x-only arithmetics. We use formulas first published by Brier and Joye[3] (see also section 3.4 of [4]). Suppose that we are working with a short Weierstraß curve of equation $y^2 = x^3 + ax + b$ for two constants a and b, and we want to compute the x coordinates of points $P_i = P_0 + iM$ for successive integers i and some base points P_0 and M. If we have the x coordinates of P_i and P_{i+1} as fractions X_i/Z_i and X_{i+1}/Z_{i+1}, respectively, and the x coordinate of M is x_m, then we can compute the x coordinate of P_{i+2} as X_{i+2}/Z_{i+2} with:

$$\begin{aligned} X_{i+2} &= Z_i((X_{i+1}x_m - aZ_{i+1})^2 - 4b(X_{i+1} + x_mZ_{i+1})Z_{i+1}) \\ Z_{i+2} &= X_i(X_{i+1} - x_mZ_{i+1})^2 \end{aligned}$$

On curve P-256, the constant b is large, but $a = -3$, making multiplications by a cheap. Overall cost is 6M+2S, which compares quite favourably with the 14M cost of the generic point addition formulas that we otherwise use in our implementation (using projective coordinates, the formulas are explained in [8]). However, the formulas above are not complete; they have a few special cases that must be handled explicitly:

- If $M = \mathbb{O}$ (the "point-at-infinity"), then there is no defined x coordinate x_m. However, all P_i are then identical to each other, making the process very simple.
- If $P_{i+1} = \mathbb{O}$, then $P_i = -M$ and $P_{i+2} = M$: we can set $X_{i+2} = x_m$ and $Z_{i+2} = 1$ in that case.
- If $P_i = \mathbb{O}$, then $P_{i+1} = M$ and $P_{i+2} = 2M$; we can then obtain the coordinates of P_{i+2} with the point doubling formulas: $X_{i+2} = (x_m-a)^2-8bx_m$, and $Z_{i+2} = 4(x_m^3+ax_m+b)$.
- If $P_i = (0, \pm\sqrt{b})$, i.e. P_i happens to be one of the two points of P-256 whose x coordinate is zero, then the following alternate formulas should be used:

$$\begin{aligned} X_{i+2} &= 2((X_{i+1}x_m + aZ_{i+1})(X_{i+1} + x_mZ_{i+1}) + 2bZ_{i+1}^2) \\ Z_{i+2} &= (X_{i+1} - x_mZ_{i+1})^2 \end{aligned}$$

In all other cases, the formulas return a proper result. In particular, when $P_{i+1} = -M$, the value $Z_{i+2} = 0$, which is correct since $P_{i+2} = \mathbb{O}$ in such a case.

The special cases can be handled with simple conditional tests, since we are working with public data, and there is no secret information that may be leaked.

The X_i/Z_i fractions can be normalized to affine x coordinates by batches, to use the batch inversion algorithm, just like EdDSA.

Performance and Comparison with EdDSA. The verification process of a truncated ECDSA signature over curve P-256 is substantially slower than for a truncated Ed25519 signature with the same number of removed bits, for the following reasons:

- Ed25519 signature truncation has "free bits" since the top three bits of s are always zero, and the fourth top bit is almost always zero as well. Since the asymptotic cost of reconstruction based on such discrete logarithms works in $O(2^{t/2})$, these "free bits" should yield an improvement by a factor of 4.
- In ECDSA, one bit is gained by the fact that both (r, s) and $(r, -s)$ are valid signatures, but that bit is lost again for the same reason: s can be negated because the value r does not distinguish between R and $-R$, but not knowing the exact point R forces us to cover both cases, which prevents us from using the same kind of optimization as in EdDSA (in EdDSA we could "center" s_1 on 2^{251}, which gained the equivalent of one extra bit, but in ECDSA we cannot, to account for the unknown sign of R).
- Contrary to the EdDSA case, precomputations do not apply because the s value is applied to the point R, which depends on the signature value, and not on the fixed generator point in the curve. Thus, we have to recompute the U_j in addition to the V_i, which leads us to use a smaller J range, and a higher I range than we could otherwise do. The prevention of precomputations explains an extra factor of about 3 between the EdDSA and ECDSA cases.
- Curve25519 is a faster curve than P-256. Multiplications and squarings in the Curve25519 field benefit from the special format of the modulus ($2^{255} - 19$) and are about 20% to 30% faster than the same operations in the P-256 field in our implementation. Moreover, formulas on P-256 involve more multiplications and squarings than their counterparts on Curve25519.

From these various slowdown factors, we expect verification of truncated ECDSA signatures to have a cost of about 15 to 20 times that of EdDSA for the same number t of removed bits. This is corroborated by benchmarks. In table 2, we give measures on the same test machines as the Ed25519 benchmarks of table 1.

As seen in the benchmarks, the verification cost of truncated ECDSA signatures can become quite high as t grows. However, even for the largest supported trunction ($t = 32$ bits, i.e. reduction of the signature by 4 full bytes), the verification time is "only" about 200 to 300 times the cost of verifying an untruncated signature. On the test ARM Cortex A53, which is the least performant of our two test systems, the worst case time (for an invalid signature) is still less than 0.3 seconds, thus compatible with many applications that run in "human time".

Acknowledgements

We thank Elena Bakos Lang and Kevin Henry for proofreading this note.

Operation	x86-64 (Intel "Coffe Lake")		aarch64 (ARM Cortex A53)	
	valid signature	invalid signature	valid signature	invalid signature
Sign	124916	-	389117	-
Verify	256582	256582	990943	990943
Verify trunc ($t = 8$)	715534	709033	2622643	2597251
Verify trunc ($t = 16$)	910975	1027641	3231167	3548516
Verify trunc ($t = 24$)	4886746	7163237	14977838	21032242
Verify trunc ($t = 28$)	18421302	27847149	53483430	79182671
Verify trunc ($t = 32$)	74496480	115365970	202775264	317982992

Table 2: Performance of ECDSA/P-256 truncated signatures. Values are in clock cycles. Signature verification measures are averages for either valid signatures, or invalid signatures.

References

1. A. Antipa, D. Brown, R. Gallant, R. Lambert, R. Struik and S. Vanstone, *Accelerated Verification of ECDSA signatures*, Selected Areas in Cryptography - SAC 2005, Lecture Notes in Computer Science, vol 3897, pp. 307-318, 2005.
2. D. Boneh, B. Lynn and H. Shacham, *Short signatures from the Weil pairing*, Advances in Cryptology - ASIACRYPT 2001, Lecture Notes in Computer Science, vol. 2248, pp. 514-532, 2001.
3. E. Brier and M. Joye, *Weierstraß Elliptic Curves and Side-Channel Attacks*, Advances in Cryptology - PKC 2002, Lecture Notes in Computer Science, vol. 2274, pp. 335-345, 2002.
4. C. Costello and B. Smith, *Montgomery curves and their arithmetic: The case of large characteristic fields*, https://eprint.iacr.org/2017/212
5. S. Josefsson and I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, https://tools.ietf.org/html/rfc8032
6. G. Neven, N. P. Smart and B.Warinschi, *Hash function requirements for Schnorr signatures*, Journal of Mathematical Cryptology, vol .3, issue 1, pp. 69-87, 2009.
7. T. Pornin, *Optimized Lattice Basis Reduction In Dimension 2, and Fast Schnorr and EdDSA Signature Verification*, https://eprint.iacr.org/2020/454
8. J. Renes, C. Costello and L. Batina, *Complete addition formulas for prime order elliptic curves*, Advances in Cryptology – Eurocrypt 2016, Lecture Notes in Computer Science, vol. 9665, pp. 403-428, 2016.
9. Y. Sakemi, T. Kobayashi, T. Saito and R. Wahby, *Pairing-Friendly Curves*, https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves