High-speed high-security signatures

Daniel J. Bernstein[1], Niels Duif[2], Tanja Lange[2],
Peter Schwabe[3], and Bo-Yin Yang[4]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to

[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
nielsduif@hotmail.com, tanja@hyperelliptic.org

[3] Department of Electrical Engineering
National Taiwan University
1, Section 4, Roosevelt Road, Taipei 10617, Taiwan
peter@cryptojedi.org

[4] Institute of Information Science
Academia Sinica, 128 Section 2 Academia Road, Taipei 115-29, Taiwan
by@crypto.tw

Abstract. This paper shows that a $390 mass-market quad-core 2.4GHz Intel Westmere (Xeon E5620) CPU can create 108000 signatures per second and verify 71000 signatures per second on an elliptic curve at a 2^{128} security level. Public keys are 32 bytes, and signatures are 64 bytes. These performance figures include strong defenses against software side-channel attacks: there is no data flow from secret keys to array indices, and there is no data flow from secret keys to branch conditions.

Keywords: Elliptic curves, Edwards curves, signatures, speed, software side channels, foolproof session keys

1 Introduction

This paper introduces software for public-key signatures with several attractive features:

– **Fast single-signature verification.** The software takes only 280880 cycles to verify a signature on Intel's widely deployed Nehalem/Westmere lines of CPUs. (This performance measurement is for short messages; for very long messages, verification time is dominated by hashing time.) Nehalem and

This work was supported by the National Science Foundation under grant 1018836, by the European Commission under Contract ICT-2007-216676 ECRYPT II, and by the National Science Council, National Taiwan University and Intel Corporation under Grant NSC99-2911-I-002-001 and 99-2218-E-001-007, and the Academia Sinica Career Award. Part of this work was carried out when Peter Schwabe was employed by Academia Sinica, Taiwan. Part of this work was carried out when Niels Duif was employed by Compumatica secure networks BV, the Netherlands. Permanent ID of this document: a1a62a2f76d23f65d622484ddd09caf8. Date: 2011.07.04.

Westmere include all Core i7, i5, and i3 CPUs released between 2008 and 2010, and most Xeon CPUs released in the same period.
- **Even faster batch verification.** The software performs a batch of 64 separate signature verifications (verifying 64 signatures of 64 messages under 64 public keys) in only 8.55 million cycles, i.e., under 134000 cycles per signature. The software fits easily into L1 cache, so contention between cores is negligible: a quad-core 2.4GHz Westmere verifies 71000 signatures per second, while keeping the maximum verification latency below 4 milliseconds.
- **Very fast signing.** The software takes only 88328 cycles to sign a message. A quad-core 2.4GHz Westmere signs 108000 messages per second.
- **Fast key generation.** Key generation is almost as fast as signing. There is a slight penalty for key generation to obtain a secure random number from the operating system; `/dev/urandom` under Linux costs about 6000 cycles.
- **High security level.** All known attacks take at least 2^{128} operations. This is the security level achieved by AES-128, NIST P-256, RSA with $\approx$ 3000-bit keys, etc. The same techniques would also produce speed improvements at other security levels.
- **Foolproof session keys.** Signatures in this paper are generated deterministically; key generation consumes new randomness but new signatures do not. This is not only a speed feature but also a security feature, directly relevant to the recent collapse of the Sony PlayStation 3 security system. See Section 2 for further discussion.
- **Collision resilience.** Hash-function collisions do not break this system. This adds a layer of defense against the possibility of weakness in the selected hash function.
- **No secret array indices.** The software never reads or writes data from secret addresses in RAM; the pattern of addresses is completely predictable. The software is therefore immune to cache-timing attacks, hyperthreading attacks, and other side-channel attacks that rely on leakage of addresses through the CPU cache.
- **No secret branch conditions.** The software never performs conditional branches based on secret data; the pattern of jumps is completely predictable. The software is therefore immune to side-channel attacks that rely on leakage of information through the branch-prediction unit.
- **Small signatures.** Signatures fit into 64 bytes. These signatures are actually compressed versions of longer signatures; the times for compression and decompression are included in the cycle counts reported above.
- **Small keys.** Public keys consume only 32 bytes. The times for compression and decompression are again included.

We have submitted our software to the eBATS project [9] for public benchmarking, and placed the software into the public domain to maximize reusability. The numbers 88328 and 280880 shown above are from the eBATS reports for our software on a Westmere CPU (Intel Xeon E5620, `hydra2`).

Our signatures are elliptic-curve signatures, carefully engineered at several levels of design and implementation to achieve very high speeds without compromising security. Section 2 specifies the signature system; Section 3 explains the techniques we use for finite-field arithmetic; Section 4 discusses fast signatures; Section 5 discusses fast verification.

Comparison to previous ECC work. Carrying out high-security elliptic-curve signature verification in only 134000 cycles on a single core of a typical Intel CPU is unprecedented. The following paragraphs discuss previous work.

Readers should be aware of several difficulties in comparing ECC performance results. First, most papers on fast ECC have been limited to ECDH (variable-base-point single-scalar multiplication) and have not implemented ECC signature verification, although there are certainly some exceptions — for example, [12] reported verification 1.33× slower than ECDH, and [22] reported verification 1.36× slower than ECDH. Second, most implementations use secret array indices and secret branch conditions and therefore must be assumed to be breakable by side-channel attacks, as illustrated by the successful OpenSSL attack in [14]; this is not an issue for public-key signature verification but it is an issue for signing and for ECDH. Third, most papers report results for only a few CPUs, so anyone without access to the same CPUs must engage in error-prone extrapolation from one CPU to another; this is not an issue for systems included in the eBATS benchmarks, but we are aware of two recent ECC implementations (discussed below) that are not included in eBATS.

Before this paper, the closest system to ours in eBATS was `ecdonaldp256`: ECDSA signatures using the NIST P-256 elliptic curve. On `hydra2` this system takes 1690936 cycles for key generation, 1790936 cycles for signing, and 2087500 cycles for verification. Better speeds were reported for ECDH: third place was `curve25519`, an implementation by Gaudry and Thomé [23] of Bernstein's Curve25519 [6]; second place was 307180 cycles for `ecfp256e`, an implementation by Hisil [27] of ECDH on an Edwards curve with similar security properties to Curve25519; and first place was 278256 cycles for `gls1271`, an implementation by Galbraith, Lin, and Scott [22] of ECDH on an Edwards curve with an endomorphism. The recent papers [26] and [29] point out security problems with endomorphisms in some ECC-based protocols, but as far as we can tell those security issues are not relevant to ECDH with standard hashing of the ECDH output, and are not relevant to ECC signatures.

Longa and Gebotys in [34] reported 281000 cycles on a Core 2 Duo E6750 for ECDH on a curve similar to `ecfp256e`, and 229000 cycles for ECDH on a curve similar to `gls1271`. The software in [34] is not included in the eBATS benchmarks and apparently is not publicly available, so we are unable to benchmark it on a Westmere. More recently Käsper in [30] reported 457813 cycles for side-channel-protected ECDH on the NIST P-224 curve on a Core 2 Duo E8400; this software is not in eBATS but has been integrated into OpenSSL.

To aid comparisons we also implemented ECDH, specifically `curve25519`, with the same side-channel defenses as our signature software (no secret array indices, and no secret branch conditions). We submitted our ECDH software to eBATS, which reports that the software uses 226872 cycles on `hydra2` for variable-base-point single-scalar multiplication. This is a new speed record for public ECDH software, a new speed record for side-channel-protected ECDH (out of all the papers mentioned above, the only ones that report side-channel protection are [6] and [30]), and a new speed record for ECDH without endomorphisms. It is even slightly better than the speed in [34] for non-side-channel-protected ECDH with endomorphisms.

Given this ECDH speed, given the ECDH-to-verification slowdowns reported in [12] and [22], and given the extra costs that we incur for decompressing keys and signatures, one would expect a verification speed close to 400000 cycles. We do better than this for several reasons, the most important reason being our use of batching. This requires careful design of the signature system, as discussed later in this paper: ECDSA, like DSA and most other signature systems, is incompatible with fast batch verification.

Comparison to other signature systems. The eBATS benchmarks cover 42 different signature systems, including various sizes of RSA, DSA, ECDSA, hyperelliptic-curve signatures, and multivariate-quadratic signatures. This paper beats almost all of the signature times and verification times (and key-generation times, which are an issue for some applications) by more than a factor of 2. The only exceptions are as follows:

- Multivariate-quadratic signatures are competitive in speed. For example, `sflashv2` takes 124740 cycles to sign and 165884 cycles to verify; `mqqsig256` takes 4216 cycles to sign and 134920 cycles to verify; smaller `mqqsig` versions are even faster. However, `sflashv2` was broken by Dubois, Fouque, Shamir, and Stern in [19]. We are not aware of any security evaluation of `mqqsig`, which was introduced last year in [24], but we disregard `mqqsig256` for the simple reason that it has a 789552-byte public key.
- `donald512` (512-bit DSA) takes 337084 cycles to verify. This is comparable to our single-signature verification speed but much slower than our batch verification speed. This is also at a far lower security level, breakable in about 2^{60} operations rather than 2^{128}.
- Some RSA-type systems provide faster verification — but this advantage decreases as the security level increases, and for many applications the advantage is outweighed by much slower signatures and much larger keys. For example, `rwb0fuz1024` (1024-bit Rabin–Williams) uses 12304 cycles to verify but 1751284 cycles to sign and 128 bytes for a public key; `ronald1024` (1024-bit RSA) uses 60628 cycles to verify but 2176212 cycles to sign and 128 bytes for a public key; `ronald3072` (3072-bit RSA) uses 230260 cycles to verify but an astonishing 31469536 cycles to sign and 384 bytes for a public key. This paper uses 134000 cycles to verify (in batches), 89416 cycles to sign, and 32 bytes for a public key.

The conventional wisdom is that RSA signatures are much better than ECC signatures in applications where each signature is verified many times, since RSA verification is much faster than ECC verification. Our ECC speed results call this conventional wisdom into question. We do not claim that our verification speeds cannot be beaten by RSA at the same security level, but we do claim that they are fast enough to make ECC an attractive option even for verification-intensive applications such as [43].

2 The signature system

This section specifies the signature system used in this paper, and a generalized signature system EdDSA that can be used with other choices of elliptic curves.

There is an extensive literature on variants of the classic signature system introduced by ElGamal in [21]; notable variants include Schnorr's signature system [44], DSA, and ECDSA. Our generalized system is another of these variants. We do not claim novelty for any of the individual modifications that we use, but we emphasize that selecting a good combination of modifications is critical for top performance. The most obvious modification is that we use twisted Edwards curves rather than Weierstrass curves; this explains our choice of the name EdDSA (Edwards-curve Digital Signature Algorithm).

EdDSA parameters. EdDSA has six parameters: an integer $b \geq 10$; a cryptographic hash function H producing $2b$-bit output; a prime power q congruent to 1 modulo 4; a $(b-1)$-bit encoding of elements of the finite field $\mathbf{F}_q$; a non-square element d of $\mathbf{F}_q$; a prime ℓ between 2^{b-4} and 2^{b-3} satisfying an extra constraint described below; and an element $B \neq (0,1)$ of the set

$$E = \{(x,y) \in \mathbf{F}_q \times \mathbf{F}_q : -x^2 + y^2 = 1 + dx^2y^2\}.$$

The condition that d is not a square implies that $d \notin \{0,-1\}$, so this set E forms a group with neutral element $0 = (0,1)$ under the twisted Edwards addition law

$$(x_1,y_1) + (x_2,y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 + x_1x_2}{1 - dx_1x_2y_1y_2}\right)$$

introduced by Bernstein, Birkner, Joye, Lange, and Peters in [7]. Completeness of the addition law — the fact that the denominators $1 \pm dx_1x_2y_1y_2$ are nonzero — follows as explained in [7, Section 6]: -1 is a square in $\mathbf{F}_q$ (since q is congruent to 1 modulo 4), so this addition law on E is $\mathbf{F}_q$-isomorphic to the Edwards addition law on the Edwards curve $x^2 + y^2 = 1 - dx^2y^2$, which is complete by [8, Theorem 3.3] since $-d$ is not a square in $\mathbf{F}_q$. The latter follows from d being a non-square and -1 being a square in $\mathbf{F}_q$. The extra constraint mentioned above is that $\ell B = 0$, where nB means the nth multiple of B in this group.

We use the encoding of $\mathbf{F}_q$ to define some field elements as being negative: specifically, x is negative if the $(b-1)$-bit encoding of x is lexicographically larger than the $(b-1)$-bit encoding of $-x$. If q is an odd prime and the encoding is the little-endian representation of $\{0, 1, \ldots, q-1\}$ then the negative elements of $\mathbf{F}_q$ are $\{1, 3, 5, \ldots, q-2\}$.

An element $(x,y) \in E$ is encoded as a b-bit string $\underline{(x,y)}$, namely the $(b-1)$-bit encoding of y followed by a sign bit; the sign bit is 1 iff x is negative. This encoding immediately determines y, and it determines x via the equation $x = \pm\sqrt{(y^2-1)/(dy^2+1)}$.

EdDSA keys and signatures. An EdDSA secret key is a b-bit string k. The hash $H(k) = (h_0, h_1, \ldots, h_{2b-1})$ determines an integer

$$a = 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i \in \{2^{b-2}, 2^{b-2} + 8, \ldots, 2^{b-1} - 8\},$$

which in turn determines the multiple $A = aB$. The corresponding EdDSA public key is $\underline{A}$. Bits $h_b, \ldots, h_{2b-1}$ of the hash are used as part of signing, as discussed in a moment.

The signature of a message M under this secret key k is defined as follows. Define $r = H(h_b, \ldots, h_{2b-1}, M) \in \{0, 1, \ldots, 2^{2b} - 1\}$; here we interpret $2b$-bit strings in little-endian form as integers in $\{0, 1, \ldots, 2^{2b} - 1\}$. Define $R = rB$. Define $S = (r + H(\underline{R}, \underline{A}, M)a) \bmod \ell$. The signature of M under k is then the $2b$-bit string $(\underline{R}, \underline{S})$, where $\underline{S}$ is the b-bit little-endian encoding of S. Applications wishing to pack data into every last nook and cranny should note that the last three bits of signatures are always 0 because ℓ fits into $b - 3$ bits.

Verification of an alleged signature on a message M under a public key works as follows. The verifier parses the key as $\underline{A}$ for some $A \in E$, and parses the alleged signature as $(\underline{R}, \underline{S})$ for some $R \in E$ and $S \in \{0, 1, \ldots, \ell - 1\}$. The verifier computes $H(\underline{R}, \underline{A}, M)$ and then checks the group equation $8SB = 8R + 8H(\underline{R}, \underline{A}, M)A$ in E. The verifier rejects the alleged signature if the parsing fails or if the group equation does not hold.

To see that signatures pass verification, simply multiply B by the equation $S = (r + H(\underline{R}, \underline{A}, M)a) \bmod \ell$, and use the fact that $\ell B = 0$, to see that $SB = rB + H(\underline{R}, \underline{A}, M)aB = R + H(\underline{R}, \underline{A}, M)A$. The verifier is *permitted* to check this stronger equation and to reject alleged signatures where the stronger equation does not hold. However, this is not *required*; checking that $8SB = 8R + 8H(\underline{R}, \underline{A}, M)A$ is enough for security.

Weak keys. Forgeries are trivial if A is a known multiple of B. For example, an attacker who knows that $A = 37B$ can choose r and compute $S = (r + 37H(\underline{R}, \underline{A}, M)) \bmod \ell$. As an even more extreme example, an attacker who knows that $A = 0B$ can choose r and compute $S = r \bmod \ell$, independently of M. We could declare that $0B$ and $37B$ are "broken" by these two "attacks" and that users must check for, and reject, these "weak keys"; but the same confused logic would require rejecting *all* keys in *all* cryptosystems, and would have no relevance to the standard definition of signature security.

Legitimate users choose $A = aB$, where a is a random secret; the derivation of a from $H(k)$ ensures adequate randomness. These users have negligible chance of generating any particular multiple of B targeted by the attacker (and no chance of generating $0B$). The chance of the attacker randomly guessing a is far smaller than the chance of the attacker computing a by known discrete-logarithm algorithms; standard elliptic-curve security criteria are designed so that the latter algorithms have negligible chance of succeeding in any reasonable amount of time.

Malleability. We also see no relevance of "malleability" to the standard definition of signature security. For example, if we slightly modified the system then replacing S by $-S$ and replacing A by $-A$ (a slight variant of the "attack" of [45]) would convert one valid signature into another valid signature of the same message under a new public key; but it would still not accomplish the attacker's goal, namely to forge a signature on a new message under a target public key. One such modification would be to omit $\underline{A}$ from the hashing; another such modification would be to have $\underline{A}$ encode only $|A|$, rather than A.

Choice of curve. Our recommended curve for EdDSA is a twisted Edwards curve birationally equivalent to the curve Curve25519 from [6]. Any efficiently computable birational equivalence preserves ECDLP difficulty, so the well-known difficulty of computing ECDLP for Curve25519 immediately implies the difficulty of computing ECDLP for our curve. We use the name Ed25519 for EdDSA with this particular choice of curve.

Specifically, Ed25519 is EdDSA with the following parameters: $b = 256$; H is SHA-512; q is the prime $2^{255} - 19$; the 255-bit encoding of $\mathbf{F}_{2^{255}-19}$ is the usual little-endian encoding of $\{0, 1, \ldots, 2^{255} - 20\}$; ℓ is the prime $2^{252} + 27742317777372353535851937790883648493$ from [6]; $d = -121665/121666 \in \mathbf{F}_q$; and B is the unique point $(x, 4/5) \in E$ for which x is positive.

Curve25519 from [6] is the Montgomery curve $v^2 = u^3 + 486662u^2 + u$ over the same field $\mathbf{F}_q$. Bernstein and Lange pointed out in [8, Section 2] that Curve25519 is birationally equivalent to an Edwards curve, specifically $x^2 + y^2 = 1 + (121665/121666)x^2y^2$; the equivalence is $x = \sqrt{486664}u/v$ and $y = (u - 1)/(u + 1)$. As above this Edwards curve is isomorphic to $-x^2 + y^2 = 1 - (121665/121666)x^2y^2$ since -1 is a square in $\mathbf{F}_q$. Our choice of base point B corresponds to the choice $u = 9$ made in [6].

Pseudorandom generation of r. ECDSA, like many other signature systems, asks users to generate not merely a random long-term secret key, but also a new random secret session key r for each message to be signed. If r becomes public then, assuming $H(\underline{R}, \underline{A}, M) \bmod \ell \neq 0$, the long-term secret key a can be simply computed as $a = (S - r)/H(\underline{R}, \underline{A}, M) \bmod \ell$. If the same value r is ever used for 2 different messages the secret key can be computed as well, as ElGamal noted in [21]. It was reported in [15] that the latter failure had occurred in Sony's ECDSA implementation for code-signing for the PlayStation3, immediately revealing Sony's long-term secret key.

Furthermore, it is well known that ECDSA's session keys are much less tolerant than the long-term key of slight deviations from randomness, even if the session keys are not revealed or reused. For example, Nguyen and Shparlinski in [40] presented an algorithm using lattice methods to compute the long-term ECDSA key from the knowledge of as few as 3 bits of r for hundreds of signatures, whether this knowledge is gained from side-channel attacks or from non-uniformity of the distribution from which r is taken.

EdDSA avoids these issues by generating $r = H(h_b, \ldots, h_{2b-1}, M)$, so that different messages will lead to different, hard-to-predict values of r. No per-message randomness is consumed. Standard PRF hypotheses imply that this session key r is indistinguishable from a truly random string generated independently for each M, so there is no loss of security. This idea of generating random signatures in a secretly deterministic way, in particular obtaining pseudorandomness by hashing a long-term secret key together with the input message, was proposed by Barwood in [3]; independently by Wigley in [47]; a few months later in a patent application [36] by Naccache, M'Raïhi, and Levy-dit-Vehel; later by

M'Raïhi, Naccache, Pointcheval, and Vaudenay in [35]; and much later by Katz and Wang in [31]. The patent application was abandoned in 2003.

EdDSA samples r from the interval $[0, 2^{2b} - 1]$, ensuring almost uniformity of the distribution modulo ℓ. The guideline [1, Section 4.1.1, Algorithm 2] specifies that the interval should be of size at least $[0, 2^{b+61} - 1]$, i.e., 64 bits more than ℓ; for Ed25519 there are 259 extra bits.

Comparison to previous ElGamal variants. The ElGamal signature system works as follows: generate a random rB for each message to be signed, and compute the signature (X, S), where X is the x-coordinate of $R = rB$ and $S = r^{-1}(H(M) + Xa) \bmod \ell$. The verifier can compute $R = S^{-1}H(M)B + S^{-1}XA$ using the public key $A = aB$ and can then verify that $X = x(R)$. (We disregard the possibility $S = 0$, which has negligible chance of occurring even under adversarial input; ECDSA is defined to check for this possibility and generate a new r, but sensible implementations will skip that check.) ElGamal's system actually uses the multiplicative group $\mathbf{F}_q^*$ with non-prime $\ell = q - 1$; ECDSA uses an elliptic-curve group with prime ℓ.

Schnorr in [44] replaced ElGamal's equation $S = r^{-1}(H(M) + x(R)a) \bmod \ell$ with the equation $S = (r + H(\underline{R}, M)a) \bmod \ell$. Schnorr's system has two attractive features:

- No inversions. This is an obvious advantage, saving time and reducing code size both for the signer and for the verifier.
- Collision resilience. The presence of $\underline{R}$ in the hash means that the attacker cannot break Schnorr's system by merely finding hash collisions.

Practical use of Schnorr's system was hampered by a patent (which expired in 2008), but the system became well known to theoreticians: the hashing of $\underline{R}$ allowed a proof (using the "forking lemma") that breaking Schnorr's system is as difficult "in the random-oracle model" as breaking DLP. See, for example, [42], [5], and [39]. We do not mean to exaggerate the real-world relevance of "provable security", but we find it obvious that Schnorr's system is a conservative, well-studied signature system.

Schnorr's signatures were not exactly (R, S): Schnorr, like ElGamal, compressed R to the hash $H(\underline{R}, M)$. The verifier can undo this compression by computing R as $SB - H(\underline{R}, M)A$. Note that this compression is public, so it cannot affect security. Neven, Smart, and Warinschi in [39] proposed taking advantage of collision resilience by choosing H to output only $b/2$ bits, reducing the size of compressed signatures by 25%; but the same proposal had actually appeared twenty years earlier in Schnorr's original paper. See [44, Section 2]. Compression of R to a hash had a much larger effect in ElGamal's original system: the system used b bits of output from H (and could not use fewer, because it was not collision-resilient), but the system used multiplicative groups rather than elliptic curves, so R needed many more than b bits. The same compression also appears in ECDSA but has no benefit there: ECDSA's hash is the same size as $\underline{R}$.

Our verification equation is the same as Schnorr's verification equation with double-size hashing instead of half-size hashing, with $\underline{A}$ inserted as an extra hash input, and *without* the compression described in the previous paragraph. These modifications obviously do not compromise security. The use of double-size hashing helps alleviate concerns regarding hash-function security; the use of $\underline{A}$ is an inexpensive way to alleviate concerns that several public keys could be attacked simultaneously; and the avoidance of compression allows an important verification speedup, as discussed in Section 5. We also reuse the double-size hash to alleviate concerns regarding nonce randomness, as discussed above.

3 Fast arithmetic modulo $2^{255} - 19$

This section explains how our software represents elements of the field $\mathbf{F}_{2^{255}-19}$, and how our software performs efficient field arithmetic. The machine instructions used in the software are available on all 64-bit Intel and AMD CPUs, but we target Intel's Nehalem/Westmere CPUs.

Multipliers on Nehalem CPUs. Field multiplications (and squarings) are the main bottlenecks in elliptic-curve performance on most CPUs. The most important tool for fast field multiplication is a fast CPU multiplication instruction. Nehalem CPUs offer three different multiplication instructions that can be used to implement high-speed field arithmetic:

- The `mulpd` instruction can perform two double-precision floating-point multiplications in SIMD fashion every cycle. Newer Sandy Bridge CPUs include a `vmulpd` instruction that can perform up to 4 double-precision floating-point multiplications per cycle, but this instruction is not available on our target CPUs.
- The `mul` instruction can multiply two 64-bit unsigned integers, producing a 128-bit result, every two cycles.
- The `pmuldq`/`pmuludq` instructions can perform two multiplications of 32-bit integers, producing 64-bit results, every cycle. The `pmuldq` instruction performs signed multiplication; the `pmuludq` instruction performs unsigned multiplication.

Multiplication and Edwards-curve arithmetic involve data-level parallelism that we could exploit with `mulpd` and `pmuldq`, but this approach would incur a serious overhead of shuffle instructions needed to arrange data in registers as described in, e.g., [17] and [38]. This overhead is eliminated when several independent computations are run in parallel, but two 64-bit results every cycle are not fundamentally better than one 128-bit result every two cycles. We therefore decompose field multiplication into multiplications of 64-bit unsigned integers.

Radix-2^{64} representation. The standard way to split 255-bit values into 64-bit limbs is a 4-limb, radix-2^{64} representation. Each element x of the field is represented as (x_0, x_1, x_2, x_3) with $x = \sum_{i=0}^{3} x_i 2^{64i}$. The multiplication of two elements x and y is decomposed into 16 multiplications of 64-bit unsigned integers; the 128-bit results are added up to produce the result in 8 limbs $r_0, \ldots, r_7$.

Reduction modulo $2^{255} - 19$ exploits the fact that $2^{256} \equiv 38$, so $38 \cdot r_4$ is added to r_0, $38 \cdot r_5$ to r_1 and so on.

A detail worth noting of this representation is that it uses 256 bits to represent 255-bit field elements. We use this one extra bit and do not always reduce modulo $2^{255} - 19$ but modulo $2^{256} - 38$. For a similar representation this has been shown to be useful for example in [10].

Our implementation of the signature scheme based on this representation of field elements yields high performance on many microprocessors such as AMD K10 or 65-nm Intel Core 2 processors. However, on our target platform, the Intel Nehalem/Westmere CPUs, this representation triggers a serious bottleneck. Every 128-bit result of the `mul` instruction is produced in two 64-bit registers. Adding two of these results requires two addition instructions. In the field multiplication most of these additions produce carries; the carry bits need to be handled by subsequent additions. The Intel Nehalem and Westmere CPUs can perform three additions per cycle, but only if these additions do not have to handle a carry bit from a previous addition (`add` instruction). An add with carry (`adc` instruction) can only be done once every two cycles; i.e., carry bits decrease addition throughput by a factor of 6. This bottleneck is triggered not only inside field multiplication and squaring but also inside additions.

Radix-2^{51} representation. To reduce the number of expensive `adc`/`subc` instructions, we instead represent an element x of $\mathbf{F}_{2^{255}-19}$ as $(x_0, x_1, x_2, x_3, x_4)$ with $x = \sum_{i=0}^{4} x_i 2^{51i}$.

The 5 limbs are unsigned integers. We can represent each element of the field $\mathbf{F}_{2^{255}-19}$ with each $x_i \in [0, \dots, 2^{51} - 1]$. In fact our implementation does not enforce these bounds except for comparisons. Multiplication accepts inputs with each limb having up to 54 bits and produces results of which each limb can be only slightly larger than 2^{51}.

Multiplication and squaring. Schoolbook multiplication of two field elements x and y, each represented in 5 unsigned integers, takes 25 `mul` instructions. The results are again produced in two 64-bit integer registers, but as both inputs have only up to 54 bits, the value in the upper result register has only up to 44 bits. Adding two multiplication results now takes only one `adc` and one `add` instruction. Furthermore reduction can be carried out simultaneously to multiplication. For example, we do not compute a coefficient r_5. Whenever the result of a `mul` instruction belongs to r_5, for example in the multiplication of $x_2 \cdot y_3$, we multiply one of the inputs by 19 and add the result to r_0. Similarly we do not compute r_6, r_7, r_8 and r_9 but directly add into $r_1, \dots, r_4$. Multiplying one input by 19 yields a result with less than 64 bits so we can use the faster `imul` instruction for these multiplications. The 5 result coefficients require 10 64-bit registers; the AMD64 architecture has 15 such registers, so we can keep the result coefficients inside registers throughout the computation.

After the multiplication we need to reduce (carry) the 5 coefficients to obtain a result with coefficients that are at most slightly larger than 2^{51}. Denote the two registers holding coefficient r_0 as r_{00} and r_{01} with $r_0 = 2^{64} r_{01} + r_{00}$. Similarly denote the two registers holding coefficient r_1 as r_{10} and r_{11}. We first shift r_{01}

left by 13, while shifting in the most significant bits of r_{00} (`shld` instruction) and then compute the logical and of r_{00} with $2^{51}-1$. We do the same with r_{10} and r_{11} and add r_{01} into r_{10} after the logical and with $2^{51}-1$. We proceed this way for coefficients $r_2,\ldots,r_4$; register r_{41} is multiplied by 19 before adding it to r_{00}. Now all 5 coefficients fit into 64-bit registers but are still too large to be used as input to another multiplication. We therefore carry from r_0 to r_1, from r_1 to r_2, from r_2 to r_3, from r_3 to r_4, and finally from r_4 to r_0. Each of these carries is done as one copy, one right shift by 51, one logical and with $2^{51}-1$, and one addition.

Squaring needs only 15 `mul` instructions. Some inputs are multiplied by 2; this is combined with multiplication by 19 where possible. The coefficient reduction after squaring is the same as for multiplication.

Multiplication and squaring are implemented as separate functions, but calls to these functions are used only for inversion (see below). Edwards-curve arithmetic uses inlined functions for point addition and doubling.

Addition, subtraction, and inversion. The results of additions do not have to be reduced if they are used as input to a multiplication. Long sequences of additions that let coefficients grow larger than 54 bits would be a problem but we do not have such sequences of additions. Field addition is therefore nothing but 5 integer additions without carries (`add` instruction). Subtraction is slightly more expensive because we use unsigned coefficients. Therefore we first add a multiple of q and then perform subtraction. This costs 5 `add` and 5 `sub` instructions.

Inversion is implemented as exponentiation with exponent $q-2$. It uses the same sequence of 255 squarings and 11 multiplications as [6].

4 Signing messages

Signature generation has three steps: (1) computing $r = H(h_b,\ldots,h_{2b-1},M)$; (2) computing $R = rB$; (3) computing $S = (r + H(\underline{R},\underline{A},M)a) \bmod \ell$.

Our primary concern is with short messages M, obviously the top concern for a server trying to keep up with a given volume of data; longer messages take more cycles per signature but far fewer cycles per byte. The computations of H take negligible time for short messages. The reduction modulo ℓ also takes negligible time with standard branchless techniques. For the rest of this section we focus on the main signing bottleneck, namely computing rB given r.

High-level strategy. We begin by computing the 253-bit integer $r \bmod \ell$. We then write $r \bmod \ell$ as $r_0 + 16r_1 + \cdots + 16^{63}r_{63}$ with

$$r_i \in \{-8,-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7\}.$$

For each i we look up $16^i|r_i|B$ in a precomputed table, and then conditionally negate $16^i|r_i|B$ to obtain $16^i r_i B$. Finally we compute rB as $\sum_i 16^i r_i B$.

There is nothing new in our computation at this level. Computing rB as a sum of precomputed pieces is a special case of a standard scalar-multiplication algorithm published by Pippenger in [41] (subsequently reinvented in [11] and

[33]); allowing negative coefficients is a standard tweak. The devil lies in the lower-level details — choosing the optimal radix 16, and computing $16^i r_i B$ and $\sum_i 16^i r_i B$ as efficiently as possible. These details are discussed below.

Low level, part 1: table lookups. Recall that, as a side-channel defense, we prohibit secret array indices. In particular, we cannot use $|r_i|$ as an array index. We instead load all table entries $0B, 16^i B, 2 \cdot 16^i B, 3 \cdot 16^i B, 4 \cdot 16^i B, 5 \cdot 16^i B, 6 \cdot 16^i B, 7 \cdot 16^i B, 8 \cdot 16^i B$ and use arithmetic operations, without branching, to combine the table entries into $16^i |r_i| B$. We similarly use arithmetic operations to compute $16^i r_i B$ from $16^i |r_i| B$ and $-16^i |r_i| B$.

We actually store table entries only for $i \in \{0, 2, 4, \ldots, 62\}$, at the expense of 4 elliptic-curve doublings. The table then contains $8 \cdot 32 = 256$ curve points (aside from $0B$, which is not stored). Each point is represented as three integers (see below) modulo $2^{255} - 19$. Each integer in turn is represented as five 8-byte words. Overall the table consumes 30 kilobytes of RAM.

We could instead use radix 32 or larger. Radix 32 would involve twice as many table loads (since we load all table entries), and twice as much arithmetic to combine table entries, but these costs would be outweighed by the benefit of fewer elliptic-curve additions. A more serious concern is that the table would be twice as large, consuming 60KB instead of 30KB. This is only a minor issue for a typical cryptographic speed test on our target CPUs (each Nehalem/Westmere core has its own fast 256KB L2 cache efficiently handling our sequential loads), but 30KB is clearly more attractive inside a larger application that needs to fit several different subroutines into L2 cache.

In the opposite direction, we could chop the table in half again at the expense of 8 more doublings; we could also switch to radix 8, 4, or 2. These changes would also allow reasonably fast signing on much smaller CPUs.

Low level, part 2: elliptic-curve addition. We use extended coordinates for the twisted Edwards curve $-x^2 + y^2 = 1 + dx^2y^2$, as proposed by Hisil, Wong, Carter, and Dawson in [28]. These coordinates are $(X : Y : Z : T)$ with $XY = ZT$ representing $x = X/Z$ and $y = Y/Z$. The addition formulas from [28, Section 3.1] are complete for our curve and use just 9 field multiplications to add a table entry (x_0, y_0) into $(X : Y : Z : T)$. Note that these formulas rely on the -1 in $-x^2$; this is why EdDSA uses the -1 twist.

One of the field multiplications is a multiplication by $d = -121665/121666$. We could replace this with a small number of multiplications by 121665 and 121666, as in [7, Section 6], but our current software treats d as a generic field element to save code size. We considered switching to a new curve using a small integer d (such as 646, which has a near-prime group order; note that we do not need the twist security of Curve25519), but decided that the resulting speedup was too small to justify departing from an established curve.

A different way to save a multiplication is to use the dual addition formulas from [28, Section 3.2]. However, those formulas are not complete; they would require a detailed analysis of intermediate results in our computation to see whether any of the intermediate additions could trigger any of the exceptional cases in the formulas.

Instead we represent a precomputed point (x_0, y_0) as $(y_0 - x_0, y_0 + x_0, 2dx_0y_0)$. These values depend only on x_0 and y_0 and are usually computed in the first part of addition in extended coordinates; providing them as part of the precomputation saves the multiplication by d, the multiplication x_0y_0, and 2 field additions, at the expense of increasing the storage requirements by a factor of 1.5. We comment that for hardware implementations this approach reduces the information exposed to template attacks trying to link multiple uses of the same precomputed point: all operations involving the precomputed point also involve the intermediate point. For details see [20, Section 5.1.2].

Results. Overall we spend a bit less than 1000 cycles for each iteration of our main signing loop, i.e., for one table lookup and one elliptic-curve mixed addition. We also spend about 21000 cycles to invert Z at the end of the computation. The complete signing procedure for a short message takes 88328 cycles.

5 Verifying signatures

Fast signature verification seems considerably more difficult than fast signature generation, for two reasons. First, the verifier has to recover the elliptic-curve points A and R from the compressed points $\underline{A}$ and $\underline{R}$. Second, checking $SB = R + H(\underline{R}, \underline{A}, M)A$ seems to require not merely a fixed-base scalar multiplication SB but also a much more expensive variable-base scalar multiplication $H(\underline{R}, \underline{A}, M)A$. This section explains several techniques that we use to address these problems.

Fast decompression. Recall that the encoding $\underline{R}$ of a point $R = (x, y)$ contains a straightforward encoding of y but contains only a sign bit for x. One must therefore recover x via the equation $x = \pm\sqrt{(y^2 - 1)/(dy^2 + 1)}$; note that $dy^2 + 1 \neq 0$ since $-d$ is not a square. The division and square root here seem to involve two exponentiations, about twice as expensive as the usual Weierstrass-curve decompression.

Of course, we could use Montgomery's trick to merge the two divisions involved in decompressing two points, but two square roots and a division are still more expensive than two Weierstrass-curve decompressions. We could also skip the compression and decompression for applications willing to use 64-byte keys and 96-byte signatures; but we think that 32-byte keys and 64-byte signatures are considerably more attractive.

To save time we look more closely at the standard computation of square roots in $\mathbf{F}_q$. The prime $q = 2^{255} - 19$ is congruent to 5 modulo 8, so any square $\alpha \in \mathbf{F}_q$ satisfies $\alpha^2 = \beta^4$ where $\beta = \alpha^{(q+3)/8}$, i.e., $\pm\alpha = \beta^2$. The standard computation is a single exponentiation to compute β, followed by a quick multiplication of β by $\sqrt{-1}$ if $\beta^2 = -\alpha$.

In the decompression context we are given α as a fraction u/v, where $u = y^2 - 1$ and $v = dy^2 + 1$. Instead of computing α we merge the division with the square-root computation:

$$\beta = (u/v)^{(q+3)/8} = u^{(q+3)/8}v^{q-1-(q+3)/8} = u^{(q+3)/8}v^{(7q-11)/8} = uv^3(uv^7)^{(q-5)/8}.$$

We check whether $\beta^2 = -\alpha$ by checking whether $v\beta^2 = -u$, and if so we multiply β by $\sqrt{-1}$. The entire computation of $\sqrt{u/v}$, starting from u and v, takes just a few multiplications more than a single exponentiation. In other words, Edwards-curve decompression is as inexpensive as Weierstrass-curve decompression.

Fast single-signature verification. To verify a single signature we use standard techniques for double-scalar multiplication to compute $SB - H(\underline{R}, \underline{A}, M)A$, and we then check whether the result is the same as R. (We actually check whether the encoding of the result is the same as the encoding of R, so that we can skip decompression of R.) The speed of Edwards-curve addition, especially with the -1 twist, makes these techniques particularly efficient; using the tables discussed in Section 4 does not seem to offer any advantage. This computation fits in very little space.

We have also considered the verification method suggested by Antipa, Brown, Gallant, Lambert, Struik, and Vanstone in [2], but our very efficient elliptic-curve arithmetic makes the overheads in this method — extra decompression and a Euclidean computation — much more troublesome. In the batch context discussed below, the only extra overhead of the method of [2] would be the Euclidean computation, but the benefit would also be much smaller.

Fast batch verification. For any system bottlenecked by signature verification, the problem is not to verify *one* signature at a time, but to verify many signatures as quickly as possible.

Naccache, M'Raïhi, Vaudenay, and Raphaeli in [37, Section 2.2] proposed verifying a batch of linear signature equations by verifying a random linear combination of the equations. This proposal is not directly applicable to ElGamal, DSA, Schnorr, ECDSA, et al., because all of those systems require *computing* linear functions (to compute R) rather than merely *verifying* linear functions; but if R is transmitted instead of $H(\cdots)$, as suggested in [37], then this problem disappears.

Unfortunately, the verification algorithm in [37] was quite slow: [37, Table 1] reported "$29n$" multiplications to verify n signatures from the same signer at a highly questionable 2^{20} security level. If the same technique were adapted to ECDSA and increased to a 2^{128} security level then it would require nearly 200 elliptic-curve additions for each signature from the same signer — somewhat faster than verifying each signature separately, but not much.

The followup paper [4] by Bellare, Garay, and Rabin proposed a more complicated verification technique using, e.g., 3200 multiplications to verify 100 exponentiations, or 6480 multiplications to verify 100 DSA signatures, in both cases at a substandard 2^{60} security level. See [4, Appendix A.1]. The number of multiplications per signature begins to drop as the batch size grows towards 1000 — see [4, Figure 3] — but such large batches do not fit into cache on typical CPUs.

The unimpressive theoretical performance of these batch-verification techniques can be traced directly to the naive exponentiation algorithms used in [37] and [4]. We do much better by using random linear combinations, as in [37], together with state-of-the-art scalar-multiplication techniques.

Specifically, we start from a batch of (M_i, A_i, R_i, S_i) where $(\underline{R_i}, \underline{S_i})$ is an alleged signature of M_i under key $\underline{A_i}$. We choose independent uniform random 128-bit integers z_i, compute $H_i = H(\underline{R_i}, \underline{A_i}, M_i)$, and verify the equation

$$\left(-\sum_i z_i S_i \bmod \ell\right)B + \sum_i z_i R_i + \sum_i (z_i H_i \bmod \ell) A_i = 0$$

by a multi-scalar multiplication. There are two reasonable choices of scalar-multiplication methods here, namely Pippenger's method in [41] and the Bos–Coster method reported in [18, Section 4]. We use the Bos–Coster method because it fits into less storage; see below for details. Note that z_i is not secret, so side-channel protection is not required.

The number of scalars here is $2n + 1$. Half of the scalars are 253-bit and half are 128-bit. If public keys appear repeatedly, the situation considered in [37] and [4], then we could save some time by merging the 253-bit scalars; this merging also explains why we do not use the similar signature equation $SB = A + H(\underline{R}, \underline{A}, M)R$, which would allow only merging 128-bit scalars. Our software focuses on general-purpose verification with arbitrary keys.

If verification succeeds then we are confident that $8S_iB = 8R_i + 8H_iA_i$ for each i, i.e., that each signature is valid. The logic is simple: the differences $P_i = 8R_i + 8H_iA_i - 8S_iB$ are elements of a cyclic group of prime order ℓ, and have been verified to satisfy $\sum_i z_i P_i = 0$; but this equation cannot hold with probability more than 2^{-128} unless all $P_i = 0$. For example, if P_4 is nonzero then the choices of $z_1, z_2, z_3, z_5, z_6, \ldots$ determine exactly one choice of z_4 satisfying $\sum_i z_i P_i = 0$, and z_4 has chance at most 2^{-128} of matching that choice.

If verification fails then there must be at least one invalid signature. We then fall back to verifying each signature separately. There are several techniques to identify a *small* number of invalid signatures in a batch, but all known techniques become slower than separate verification as the number of invalid signatures increases; separate verification provides the best defense against denial-of-service attacks.

Fast multi-scalar multiplication. The Bos–Coster method mentioned above is as follows: to compute $n_1P_1 + n_2P_2 + \cdots$, where $n_1 \geq n_2 \geq \cdots$, we recursively compute $(n_1 - n_2)P_1 + n_2(P_1 + P_2) + \cdots$. For n_1 much larger than n_2, say $2^{k+1}n_2 > n_1 \geq 2^k n_2$, we could gain speed by instead recursively computing $(n_1 - 2^k n_2)P_1 + n_2(2^k P_1 + P_2) + \cdots$, but we have found this to occur so rarely that checking for it is not worthwhile.

We keep the scalars n_i in a heap so that identifying the two largest scalars is easy. The usual method to insert a new element into a heap is top-down, starting at the root and swapping down for a variable number of steps. We instead use Floyd's 1964 bottom-up algorithm discussed in [32, Exercise 5.2.3–18] (often miscredited to [16] and [46]): start at the root, swap down to the bottom, and then swap up for a variable number of steps. This has the advantage of somewhat reducing the number of comparisons, and the not-so-well-known advantage of drastically reducing the number of branches, especially for balanced heaps.

Results. The complete verification procedure takes under 134000 cycles per signature for batch size 64. Our batch-verification software is included in, although not yet benchmarked by, the public eBATS benchmarking framework.

Doubling the batch size to 128 no longer fits into L1 cache but still improves performance on our target CPU, taking under 125000 cycles per signature. Larger batches take under 114000 cycles per signature while still fitting into L2 cache. Our software spends about 44000 cycles on decompression, so verification of uncompressed signatures (32 extra bytes) using uncompressed public keys (another 32 extra bytes) would take only about 81000 cycles for batch size 128, even faster than signing. However, in this paper we have emphasized the performance that we obtain without using so much space.

References

[1] — (no editor), *Technical guideline TR-03111, elliptic curve cryptography* (2009). Citations in this document: §2.

[2] A. Antipa, D. R. L. Brown, R. P. Gallant, R. J. Lambert, R. Struik, S. A. Vanstone, *Accelerated verification of ECDSA signatures*, in SAC 2005, LNCS 3897 (2006), 307–318. Citations in this document: §5, §5.

[3] G. Barwood, *Digital signatures using elliptic curves*, message `32f519ad.19609226@news.dial.pipex.com` posted to `sci.crypt` (1997). URL: `http://groups.google.com/group/sci.crypt/msg/b28aba37180dd6c6`. Citations in this document: §2.

[4] M. Bellare, J. A. Garay, T. Rabin, *Fast batch verification for modular exponentiation and digital signatures*, in Eurocrypt '98, LNCS 1403 (1998), 236–250. Citations in this document: §5, §5, §5, §5, §5.

[5] M. Bellare, G. Neven, *Multi-signatures in the plain public-key model and a general forking lemma*, in CCS 2006 (2006), 390–399. Citations in this document: §2.

[6] D. J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006, LNCS 3958 (2006), 207–228. Citations in this document: §1, §1, §2, §2, §2, §2, §3.

[7] D. J. Bernstein, P. Birkner, M. Joye, T. Lange, C. Peters, *Twisted Edwards curves*, in Africacrypt 2008, LNCS 5023 (2008), 389–405. Citations in this document: §2, §2, §4.

[8] D. J. Bernstein, T. Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007, LNCS 4833 (2007), 29–50. Citations in this document: §2, §2.

[9] D. J. Bernstein, T. Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 4 July 2011 (2011). URL: `http://bench.cr.yp.to/ebats.html`. Citations in this document: §1.

[10] J. W. Bos, *High-performance modular multiplication on the Cell processor*, in WAIFI 2010, LNCS 6087 (2010), 7–24. Citations in this document: §3.

[11] E. F. Brickell, D. M. Gordon, K. S. McCurley, D. B. Wilson, *Fast exponentiation with precomputation (extended abstract)*, in Eurocrypt '92, LNCS 658 (1993), 200–207. Citations in this document: §4.

[12] M. Brown, D. Hankerson, J. López, A. Menezes, *Software implementation of the NIST elliptic curves over prime fields* (2000); see also newer version [**13**]. URL: `http://www.cacr.math.uwaterloo.ca/techreports/2000/corr2000-56.ps`. Citations in this document: §1, §1.

[13] M. Brown, D. Hankerson, J. López, A. Menezes, *Software implementation of the NIST elliptic curves over prime fields*, in CT-RSA 2001, LNCS 2020 (2001), 250–265; see also older version [**12**]. MR 1907102.

[14] B. B. Brumley, R. M. Hakala, *Cache-timing template attacks*, in Asiacrypt 2009, LNCS 5912 (2009), 667–684. Citations in this document: §1.

[15] "Bushing", H. M. "marcan" Cantero, S. Boessenkool, S. Peter, *PS3 epic fail* (2010). URL: `http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf`. Citations in this document: §2.

[16] S. Carlsson, *Average-case results on heapsort*, BIT **27** (1987), 2–17. Citations in this document: §5.

[17] N. Costigan, P. Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine*, in Africacrypt 2009, LNCS 5580 (2009), 368–385. Citations in this document: §3.

[18] P. de Rooij, *Efficient exponentiation using precomputation and vector addition chains*, in Eurocrypt '94, LNCS 950 (1995), 389–399. Citations in this document: §5.

[19] V. Dubois, P.-A. Fouque, A. Shamir, J. Stern, *Practical cryptanalysis of SFLASH*, in Crypto 2007, LNCS 4622 (2007), 1–12. Citations in this document: §1.

[20] N. Duif, *Smart card implementation of a digital signature scheme for Twisted Edwards curves*, M.A. thesis, Technische Universiteit Eindhoven, 2011. Citations in this document: §4.

[21] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory **31** (1985), 469–472. Citations in this document: §2, §2.

[22] S. Galbraith, X. Lin, M. Scott, *Endomorphisms for faster elliptic curve cryptography on a large class of curves*, in Eurocrypt 2009, LNCS 5479 (2009), 518–535. Citations in this document: §1, §1, §1.

[23] P. Gaudry, E. Thomé, *The mpFq library and implementing curve-based key exchanges*, in SPEED 2007 (2007), 49–64. Citations in this document: §1.

[24] D. Gligoroski, R. S. Odegøard, R. E. Jensen, L. Perret, J.-C. Faugère, S. J. Knapskog, S. Markovski, *The digital signature scheme MQQ-SIG* (2010). Citations in this document: §1.

[25] E.-J. Goh, S. Jarecki, J. Katz, N. Wang, *Efficient signature schemes with tight reductions to the Diffie-Hellman problems*, Journal of Cryptology **20** (2007), 493–514. See [31].

[26] R. Granger, *On the static Diffie–Hellman problem on elliptic curves over extension fields*, in Asiacrypt 2010, LNCS 6477 (2010), 283–302. Citations in this document: §1.

[27] H. Hisil, *Elliptic curves, group law, and efficient computation*, Ph.D. thesis, Queensland University of Technology, 2010. Citations in this document: §1.

[28] H. Hisil, K. K.-H. Wong, G. Carter, E. Dawson, *Twisted Edwards curves revisited*, in Asiacrypt 2008, LNCS 5350 (2008), 326–343. Citations in this document: §4, §4, §4.

[29] A. Joux, V. Vitse, *Elliptic curve discrete logarithm problem over small degree extension fields. Application to the static Diffie–Hellman problem on* $E(\mathbf{F}_{q^5})$ (2010). Citations in this document: §1.

[30] E. Käsper, *Fast elliptic curve cryptography in OpenSSL*, in RLCPS 2011, to appear (2011). Citations in this document: §1, §1.

[31] J. Katz, N. Wang, *Efficiency improvements for signature schemes with tight security reductions*, in CCS 2003 (2003), 155–164; portions incorporated into [**25**]. Citations in this document: §2.

[32] D. E. Knuth, *The art of computer programming, volume 3: sorting and searching*, 2nd edition, Addison-Wesley, Reading, 1998. Citations in this document: §5.

[33] C. H. Lim, P. J. Lee, *More flexible exponentiation with precomputation*, in CRYPTO 1994, LNCS 839 (1994), 95–107. Citations in this document: §4.

[34] P. Longa, C. H. Gebotys, *Efficient techniques for high-speed elliptic curve cryptography*, in CHES 2010, LNCS 6225 (2010), 80–94. Citations in this document: §1, §1, §1.

[35] D. M'Raïhi, D. Naccache, D. Pointcheval, S. Vaudenay, *Computational alternatives to random number generators*, in SAC '98, LNCS 1556 (1999), 72–80. Citations in this document: §2.

[36] D. Naccache, D. M'Raïhi, F. Levy-dit-Vehel, *Patent application WO/1998/051038: pseudo-random generator based on a hash coding function for cryptographic systems requiring random drawing* (1997). Citations in this document: §2.

[37] D. Naccache, D. M'Raïhi, S. Vaudenay, D. Raphaeli, *Can D.S.A. be improved? Complexity trade-offs with the digital signature standard*, in Eurocrypt '94, LNCS 950 (1994). Citations in this document: §5, §5, §5, §5, §5, §5, §5.

[38] M. Naehrig, R. Niederhagen, P. Schwabe, *New software speed records for cryptographic pairings*, in Latincrypt 2010, LNCS 6212 (2010), 109–123. Citations in this document: §3.

[39] G. Neven, N. P. Smart, B. Warinschi, *Hash function requirements for Schnorr signatures*, Journal of Mathematical Cryptology **3** (2009), 69–87. Citations in this document: §2, §2.

[40] P. Q. Nguyen, I. Shparlinski, *The insecurity of the elliptic curve digital signature algorithm with partially known nonces*, Designs, Codes and Cryptography **30** (2003), 201–217. Citations in this document: §2.

[41] N. Pippenger, *On the evaluation of powers and related problems (preliminary version)*, in FOCS '76 (1976), 258–263. Citations in this document: §4, §5.

[42] D. Pointcheval, J. Stern, *Security arguments for digital signatures and blind signatures*, Journal of Cryptology **13** (2000), 361–396. Citations in this document: §2.

[43] J. Rangasamy, D. Stebila, C. Boyd, J. González Nieto, *An integrated approach to cryptographic mitigation of denial-of-service attacks*, in ASIACCS 2011 (2011). Citations in this document: §1.

[44] C. P. Schnorr, *Efficient identification and signatures for smart cards*, in Crypto '89, LNCS 435 (1990), 239–252. Citations in this document: §2, §2, §2.

[45] J. Stern, D. Pointcheval, J. Malone-Lee, N. P. Smart, *Flaws in applying proof methodologies to signature schemes*, in Crypto 2002, LNCS 2442 (2002), 93–110. Citations in this document: §2.

[46] I. Wegener, *Bottom-up-heapsort, a new variant of heapsort, beating, on average, quicksort (if n is not very small)*, Theoretical Computer Science **118** (1993), 81–98. Citations in this document: §5.

[47] J. Wigley, *Removing need for rng in signatures*, message `5gov5d$pad@wapping.ecs.soton.ac.uk` posted to `sci.crypt` (1997). URL: `http://groups.google.com/group/sci.crypt/msg/a6da45bcc8939a89`. Citations in this document: §2.