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 60607–7053, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, the 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 109000 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 273364 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

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: 2011.09.26.

messages, verification time is dominated by hashing time.) Nehalem and 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 87548 cycles to sign a message. A quad-core 2.4GHz Westmere signs 109000 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.** This system has a 2^{128} security target; breaking it has similar difficulty to breaking NIST P-256, RSA with $\approx$ 3000-bit keys, strong 128-bit block ciphers, etc. (The same techniques would also produce speed improvements at other security levels.) The best attacks known actually cost more than 2^{140} bit operations on average, and degrade quadratically in success probability as the number of bit operations drops.
- **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 [15] for public benchmarking, and placed the software into the public domain to maximize reusability. The numbers 87548 and 273364 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, [21] reported verification 1.33× slower than ECDH, and [34] 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 [23]; 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.

Intel's "Turbo Boost" and AMD's "Turbo Core" have added a further difficulty for new CPUs. Typical benchmarking frameworks measure performance on a single CPU core, and Turbo Boost fools most of these frameworks into reporting excessively low Westmere cycle counts — speeds that the CPU cannot actually achieve when a sensible workload is keeping all cores busy. The eBATS reports include explicit warnings regarding Turbo Boost. This corruption does not occur on `hydra2`: Turbo Boost is disabled on that computer.

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 [35] of Bernstein's Curve25519 [12].
- Second place was 307180 cycles for `ecfp256e`, an implementation by Hisil [40] of ECDH on an Edwards curve with similar security properties to Curve25519.
- First place was 278256 cycles for `gls1271`, an implementation by Galbraith, Lin, and Scott [34] of ECDH on an Edwards curve with an endomorphism.

The recent papers [38] and [44] 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 [51] claimed 281000 cycles on a Core 2 Duo E6750 (C2 65nm) for ECDH on a curve similar to `ecfp256e`, and 229000 cycles for ECDH on a curve similar to `gls1271`. The software in [51] 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 [46] reported 457813 cycles for side-channel-protected ECDH on the NIST P-224 curve on a Core 2 Duo E8400 (C2 45nm); this software has been integrated into OpenSSL but not yet into eBATS.

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 [12] and [46]), and a new speed record for ECDH without endomorphisms.

We do not claim that `curve25519` will maintain its current position on top of eBATS: we would expect ECDH with endomorphisms (especially without side-channel protection) to be somewhat faster than ECDH without endomorphisms on many platforms. This expectation seems to be supported by the very recent paper [42] by Hu, Longa, and Xu: [42, Table 2] claims 194000 cycles for a curve with endomorphisms on an Intel Core 2 Duo E6750, and the accompanying web site [42, reference 18] claims 182000 cycles on an Intel Core i5 540M (Westmere). The same web site also claims 215000 Westmere cycles for a curve without endomorphisms. However, like the software in [51], the software in [42] does not appear to be publicly available. The resulting lack of verifiability raises questions regarding accuracy. We are particularly skeptical of the Westmere speed claims, given the Turbo Boost issues discussed above. After we wrote this paragraph, the same web site was updated to claim 250000 cycles for the same software on another Westmere CPU.

Given our 226872-cycle ECDH speed, given the ECDH-to-verification slowdowns reported in [21] and [34], 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 4212 cycles to sign and 134900 cycles to verify; smaller `mqqsig` versions are even faster. However, `sflashv2` was broken by Dubois, Fouque, Shamir, and Stern in [30]. We are not aware of any security evaluation of `mqqsig`, which was introduced last year in [36], but we disregard `mqqsig256` for the simple reason that it has a 789552-byte public key.
– `donald512` (512-bit DSA) takes 334508 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} simple operations.
– 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 60300 cycles to verify but 2171124 cycles to sign and 128 bytes for a public key; `ronald3072` (3072-bit RSA) uses 231536 cycles to verify but an astonishing 31456912 cycles to sign and 384 bytes for a public key. This paper uses 134000 cycles to verify (in batches), 87548 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 [70].

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 [33]; notable variants include Schnorr's signature system [72], 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 seven 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

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

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 [13]. Completeness of the addition law — the fact that the denominators $1 \pm dx_1x_2y_1y_2$ are nonzero — follows as explained in [13, 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 [14, 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

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

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 $r = H(h_b, \ldots, h_{2b-1}, M) \in \left\{0,1,\ldots,2^{2b}-1\right\}$; here we interpret $2b$-bit strings in little-endian form as integers in $\left\{0,1,\ldots,2^{2b}-1\right\}$. 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 [75]) 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 [12]. 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-SHA-512 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 $2^{252} +$

2774231777737235353585193779088364849 3 from [12]; $d = -121665/121666 \in \mathbf{F}_q$; and B is the unique point $(x, 4/5) \in E$ for which x is positive.

Curve25519 from [12] is the Montgomery curve $v^2 = u^3 + 486662u^2 + u$ over the same field $\mathbf{F}_q$. Bernstein and Lange pointed out in [14, 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 [12].

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 [33, Note 2]. It was reported in [24] 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 [61] 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. 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 [9]; independently by Wigley in [79]; a few months later in a patent application [57] by Naccache, M'Raïhi, and Levy-dit-Vehel; later by M'Raïhi, Naccache, Pointcheval, and Vaudenay in [55]; and much later by Katz and Wang in [47]. The patent application was abandoned in 2003.

Standard PRF hypotheses imply that this pseudorandom session key r is indistinguishable from a truly random string generated independently for each M, so there is no loss of security. Well-known length-extension properties prevent secret-prefix SHA-512 from being a PRF, but also do not threaten the security of Ed25519-SHA-512, since r is not visible to the attacker. All remaining SHA-3 candidates are explicitly designed to be PRFs, and we will not hesitate to recommend Ed25519-SHA-3 after SHA-3 is standardized. It would of course also be safe to generate r with a cipher such as AES, combined with standard PRF-stretching mechanisms to support a long input; but we prefer to reuse H to save area in hardware implementations.

EdDSA samples r from the interval $[0, 2^{2b} - 1]$, ensuring almost uniformity of the distribution modulo ℓ. The guideline [2, 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 original ElGamal system [33, Section III] predated elliptic-curve cryptography; it instead used the multiplicative group $\mathbf{F}_q^*$. ElGamal took a large non-prime ℓ, specifically $\ell = q - 1$, and focused on the case of prime q. ElGamal's signatures were pairs (R, S) of integers between 0 and $q-2$ inclusive satisfying $B^{H(M)} = A^R R^S$ in $\mathbf{F}_q^*$. See [33, equation (3)]; see also [33, Attack 6] for the introduction of H. The signer, given M, generates a random r coprime to ℓ and computes the signature (R, S), where $R = B^r$ and $S = r^{-1}(H(M) - Ra) \bmod \ell$.

Schnorr in [72] pointed out that one could safely work in an order-ℓ subgroup of $\mathbf{F}_q^*$ with a prime ℓ much smaller than q, saving most of the space for S. Schnorr also introduced several other improvements to ElGamal's system, as discussed below.

ElGamal's verification equation involves R as an element of the group $\mathbf{F}_q^*$ and as a scalar, the exponent for A. For more general groups one needs a function x mapping group elements to scalars. ECDSA works this way: it replaces $\mathbf{F}_q^*$ with an order-ℓ subgroup of an elliptic-curve group over $\mathbf{F}_q$ and defines $x(R)$ as the x-coordinate of R. ECDSA also replaces A with $-A$, changing the signer's subtraction into an addition and obtaining the verification equation $H(M)B + x(R)A = SR$. ECDSA replaces this three-scalar equation with the equivalent two-scalar equation $S^{-1}H(M)B + S^{-1}x(R)A = R$ at the expense of requiring S to be invertible modulo ℓ; note that both the signer and the verifier compute inverses here.

Schnorr used a cryptographic hash function for x. This has minimal expense and eliminates any concerns regarding the mathematical structure of simpler functions x. Schnorr also compressed the group element R to the scalar $x(R)$: a Schnorr signature is $(x(R), S)$ rather than (R, S). Given a compressed signature $(x(R), S)$, the verifier recomputes R as $S^{-1}H(M)B + S^{-1}x(R)A$ and checks that $x(R)$ matches; at this point the verifier knows a valid uncompressed signature (R, S), so the compression cannot reduce security.

Schnorr also merged the hashing of R with the hashing of M. One way to understand this merging is to replace S with $x(R)S$, and to impose the extra constraint $x(R) \neq 0$, obtaining the verification equation $x(R)^{-1}H(M)B + A = SR$. There is no need for the multiplicative structure of $x(R)^{-1}H(M)$ here: one can instead use the verification equation $H(\underline{R}, M)B + A = SR$, with the signer obtaining S as $r^{-1}(H(\underline{R}, M) + a) \bmod \ell$. Schnorr actually used the equation $SB = R + H(\underline{R}, M)A$, eliminating all inversions both for the signer and for the verifier; this is an obvious advantage, saving time and reducing code size.

The presence of R as input to the hash function provides collision resilience: the attacker cannot break Schnorr's system by merely finding hash collisions. Neven, Smart, and Warinschi in [60] 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 [72, Section 2].

Practical use of Schnorr's system was hampered by a patent (which expired in 2008), but the system became well known to theoreticians, because the hashing of $\underline{R}$ allowed various security proofs. Some proofs use the "forking lemma" to show that any generic-hash attack against Schnorr's system (i.e., any attack that works for arbitrary functions H) can be converted into a DLP algorithm with a polynomially bounded, although often quite severe, loss of efficiency. There are also theorems with a different loss of efficiency for generic-group attacks (i.e., attacks that work for arbitrary groups) under mild assumptions on H, and theorems with no loss of efficiency for generic-group generic-hash attacks. See, for example, [67], [74], [11], and [60]. 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.

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* Schnorr's compression of R. 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., [26] and [59]. 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 [17].

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 $x_i \in [0, \ldots, 2^{51} - 1]$. 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 $r_1, \ldots, r_4$. 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 [12].

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 [64] (subsequently reinvented in [19] and [50]); 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 [41]. These coordinates are $(X : Y : Z : T)$ with

$XY = ZT$ representing $x = X/Z$ and $y = Y/Z$. The addition formulas from [41, 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 [13, 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 [41, 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.

Template attacks. We comment that for hardware implementations this type of precomputation reduces the information exposed to template attacks trying to link multiple uses of the same precomputed point.

Consider, for example, an attacker monitoring the power consumption of a device with very limited memory. Assume that the device designer has reduced the table described above to just $B, 2B, \ldots, 8B$ at the expense of many doublings, and has saved more memory by storing a table entry as simply (x_0, y_0). The addition formulas then begin by computing $y_0 - x_0$, $y_0 + x_0$, etc. If the same table entry is used again later then the same subtraction, addition, etc. will be performed again, resulting in exactly the same power trace. The attacker can therefore partition the loaded points into (at most) 16 different groups, obtaining 55 bits of information on average, as discussed in [31, Section 5.1.2].

Precomputing $y_0 - x_0$, $y_0 + x_0$, etc. guarantees (for these addition formulas) that all operations involving the precomputed point also involve the intermediate point, which varies unpredictably between different uses of the same table entry. A closer look at field arithmetic sometimes reveals lower-level operations that depend on only one input, such as the preliminary additions in Karatsuba's method; the results of those operations can be similarly precomputed.

Of course, there is much more to say about countermeasures to hardware side-channel attacks; we do not claim that any single countermeasure is adequate by itself. The software situation is simpler, since the side channels exposed to an attacker are much more limited.

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 87548 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 [7], 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 [7] 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 [58, 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, etc., 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 [58], then this problem disappears.

Unfortunately, the verification algorithm in [58] was quite slow: [58, 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 [10] 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 [10, Appendix A.1]. The number of multiplications per signature begins to drop as the batch size grows towards 1000 — see [10, 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 [58] and [10]. We do much better by using random linear combinations, as in [58], 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 [64] and the Bos–Coster method reported in [27, Section 4]. We use the Bos–Coster method be-

cause 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 [58] and [10], 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 $\sum_i z_iP_i = 0$; 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 $\sum_i z_iP_i = 0$, 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 $2^{k+1}n_2 > n_1 \geq 2^kn_2$, we could gain speed by instead recursively computing $(n_1 - 2^kn_2)P_1 + n_2(2^kP_1 + P_2) + \cdots$, 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 replace the root of 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 [48, Exercise 5.2.3–18] (often miscredited to [25] and [78]): 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), *17th annual symposium on foundations of computer science*, IEEE Computer Society, Long Beach, California, 1976. MR 56:1766. See [64].

[2] — (no editor), *Technical guideline TR-03111, elliptic curve cryptography* (2009). URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03111/BSI-TR-03111_pdf.pdf?__blob=publicationFile. Citations in this document: §2.

[3] — (no editor), *SPEED: software performance enhancement for encryption and decryption*, 2007. URL: http://www.hyperelliptic.org/SPEED. See [35].

[4] — (no editor), *Proceedings of the 6th ACM symposium on information, computer and communications security, Hong Kong, March 22–24, 2011*, Association for Computing Machinery, 2011. ISBN 978-1-4503-0564-8. See [70].

[5] Michel Abdalla, Paulo S. L. M. Barreto (editors), *Progress in cryptology — LATINCRYPT 2010, first international conference on cryptology and information security in Latin America, Puebla, Mexico, August 8–11, 2010, proceedings*, Lecture Notes in Computer Science, 6212, Springer, 2010. ISBN 978-3-642-14711-1. See [59].

[6] Masayuki Abe (editor), *Advances in cryptology — ASIACRYPT 2010, 16th international conference on the theory and application of cryptology and information security, Singapore, December 5–9, 2010, proceedings*, Lecture Notes in Computer Science, 6477, Springer, 2010. ISBN 978-3-642-17372-1. See [38].

[7] Adrian Antipa, Daniel R. L. Brown, Robert P. Gallant, Robert J. Lambert, René Struik, Scott A. Vanstone, *Accelerated verification of ECDSA signatures*, in SAC 2005 [**69**] (2006), 307–318. MR 2007d:94044. URL: http://www.cacr.math.uwaterloo.ca/techreports/2005/tech_reports2005.html. Citations in this document: §5, §5.

[8] Vijay Atluri, Trent Jaeger (program chairs), *Proceedings of the 10th ACM conference on computer and communications security*, ACM Press, 2003. ISBN 1-58113-738-9. See [47].

[9] George 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.

[10] Mihir Bellare, Juan A. Garay, Tal Rabin, *Fast batch verification for modular exponentiation and digital signatures*, in Eurocrypt '98 [**62**] (1998), 236–250. URL: http://cseweb.ucsd.edu/~mihir/papers/batch.html. Citations in this document: §5, §5, §5, §5, §5.

[11] Mihir Bellare, Gregory Neven, *Multi-signatures in the plain public-key model and a general forking lemma*, in CCS 2006 [**45**] (2006), 390–399. URL: http://cseweb.ucsd.edu/~mihir/papers/multisignatures.html. Citations in this document: §2.

[12] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in PKC 2006 [**81**] (2006), 207–228. URL: http://cr.yp.to/papers.html#curve25519. Citations in this document: §1, §1, §2, §2, §2, §2, §3.

[13] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, Christiane Peters, *Twisted Edwards curves*, in Africacrypt 2008 [**77**] (2008), 389–405. URL: http://eprint.iacr.org/2008/013. Citations in this document: §2, §2, §4.

[14] Daniel J. Bernstein, Tanja Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007 [**49**] (2007), 29–50. URL: http://eprint.iacr.org/2007/286. Citations in this document: §2, §2.

[15] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 19 September 2011 (2011). URL: http://bench.cr.yp.to. Citations in this document: §1.

[16] G. R. Blakley, David Chaum (editors), *Advances in cryptology, proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19–22, 1984, proceedings*, Lecture Notes in Computer Science, 196, Springer, Berlin, 1985. ISBN 3-540-15658-5. MR 86j:94003. See [32].

[17] Joppe W. Bos, *High-performance modular multiplication on the Cell processor*, in WAIFI 2010 [**39**] (2010), 7–24. Citations in this document: §3.

[18] Gilles Brassard (editor), *Advances in cryptology — CRYPTO '89, 9th annual international cryptology conference, Santa Barbara, California, USA, August 20–24, 1989, proceedings*, Lecture Notes in Computer Science, 435, Springer, Berlin, 1990. ISBN 3-540-97317-6. MR 91b:94002. See [72].

[19] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, David B. Wilson, *Fast exponentiation with precomputation (extended abstract)*, in Eurocrypt '92 [**71**] (1993), 200–207; see also newer version [**20**]. URL: http://cr.yp.to/bib/entries.html#1993/brickell-exp. Citations in this document: §4.

[20] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, David B. Wilson, *Fast exponentiation with precomputation: algorithms and lower bounds* (1995); see also older version [**19**]. URL: http://research.microsoft.com/~dbwilson/bgmw/.

[21] Michael Brown, Darrel Hankerson, Julio López, Alfred Menezes, *Software implementation of the NIST elliptic curves over prime fields* (2000); see also newer version [**22**]. URL: http://www.cacr.math.uwaterloo.ca/techreports/2000/corr2000-56.ps. Citations in this document: §1, §1.

[22] Michael Brown, Darrel Hankerson, Julio López, Alfred Menezes, *Software implementation of the NIST elliptic curves over prime fields*, in CT-RSA 2001 [**56**] (2001), 250–265; see also older version [**21**]. MR 1907102.

[23] Billy Bob Brumley, Risto M. Hakala, *Cache-timing template attacks*, in Asiacrypt 2009 [**53**] (2009), 667–684. Citations in this document: §1.

[24] "Bushing", Hector Martin "marcan" Cantero, Segher Boessenkool, Sven 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.

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

[26] Neil Costigan, Peter Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine*, in Africacrypt 2009 [**68**] (2009), 368–385. URL: http://cryptojedi.org/users/peter/#celldh. Citations in this document: §3.

[27] Peter de Rooij, *Efficient exponentiation using precomputation and vector addition chains*, in Eurocrypt '94 [**28**] (1995), 389–399. MR 1479665. Citations in this document: §5.

[28] Alfredo De Santis (editor), *Advances in cryptology — EUROCRYPT '94, workshop on the theory and application of cryptographic techniques, Perugia, Italy, May 9–12, 1994, proceedings*, Lecture Notes in Computer Science, 950, Springer, Berlin, 1995. ISBN 3-540-60176-7. MR 98h:94001. See [27], [58].

[29] Yvo Desmedt (editor), *Advances in cryptology — CRYPTO '94, 14th annual international cryptology conference, Santa Barbara, California, USA, August 21–25, 1994, proceedings*, Lecture Notes in Computer Science, 839, Springer, Berlin, 1994. ISBN 3-540-58333-5. See [50].

[30] Vivien Dubois, Pierre-Alain Fouque, Adi Shamir, Jacques Stern, *Practical cryptanalysis of SFLASH*, in Crypto 2007 [**54**] (2007), 1–12. URL: http://eprint.iacr.org/2007/141. Citations in this document: §1.

[31] Niels Duif, *Smart card implementation of a digital signature scheme for Twisted Edwards curves*, M.A. thesis, Technische Universiteit Eindhoven, 2011. URL: http://www.nielsduif.nl/2011_05_20_report_final.pdf. Citations in this document: §4.

[32] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, in Crypto '84 [**16**] (1985), 10–18; see also newer version [**33**]. MR 87b:94037.

[33] Taher ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory **31** (1985), 469–472; see also older version [**32**]. ISSN 0018-9448. MR 86j:94045. Citations in this document: §2, §2, §2, §2, §2.

[34] Steven Galbraith, Xibin Lin, Michael Scott, *Endomorphisms for faster elliptic curve cryptography on a large class of curves*, in Eurocrypt 2009 [**43**] (2009), 518–535. URL: http://eprint.iacr.org/2008/194. Citations in this document: §1, §1, §1.

[35] Pierrick Gaudry, Emmanuel Thomé, *The mpFq library and implementing curve-based key exchanges*, in SPEED [**3**] (2007), 49–64. URL: http://www.loria.fr/~gaudry/papers.en.html. Citations in this document: §1.

[36] Danilo Gligoroski, Rune Steinsmo Odegøard, Rune Erlend Jensen, Ludovic Perret, Jean-Charles Faugère, Svein Johan Knapskog, Smile Markovski, *The digital signature scheme MQQ-SIG* (2010). URL: http://eprint.iacr.org/2010/527.pdf. Citations in this document: §1.

[37] Eu-Jin Goh, Stanislaw Jarecki, Jonathan Katz, Nan Wang, *Efficient signature schemes with tight reductions to the Diffie-Hellman problems*, Journal of Cryptology **20** (2007), 493–514. URL: http://www.cs.umd.edu/~jkatz/papers.html. See [47].

[38] Robert Granger, *On the static Diffie–Hellman problem on elliptic curves over extension fields*, in Asiacrypt 2010 [**6**] (2010), 283–302. URL: http://eprint.iacr.org/2010/177. Citations in this document: §1.

[39] M. Anwar Hasan, Tor Helleseth (editors), *Arithmetic of finite fields, third international workshop, WAIFI 2010, Istanbul, Turkey, June 27–30, 2010, proceedings*, Lecture Notes in Computer Science, 6087, Springer, 2010. ISBN 978-3-642-13796-9. See [17].

[40] Hüseyin Hisil, *Elliptic curves, group law, and efficient computation*, Ph.D. thesis, Queensland University of Technology, 2010. URL: http://eprints.qut.edu.au/33233. Citations in this document: §1.

[41] Hüseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson, *Twisted Edwards curves revisited*, in Asiacrypt 2008 [**63**] (2008), 326–343. URL: http://eprint.iacr.org/2008/522. Citations in this document: §4, §4, §4.

[42] Zhi Hu, Patrick Longa, Maozhi Xu, *Implementing 4-dimensional GLV method on GLS elliptic curves with j-invariant 0* (2011). URL: http://eprint.iacr.org/2011/315. Citations in this document: §1, §1, §1, §1.

[43] Antoine Joux (editor), *Advances in cryptology — EUROCRYPT 2009, 28th annual international conference on the theory and applications of cryptographic techniques, Cologne, Germany, April 26–30, 2009, proceedings*, Lecture Notes in Computer Science, 5479, Springer, 2009. ISBN 978-3-642-01000-2. See [34].

[44] Antoine Joux, Vanessa 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). URL: http://eprint.iacr.org/2010/157. Citations in this document: §1.

[45] Ari Juels, Rebecca N. Wright, Sabrina De Capitani di Vimercati (editors), *Proceedings of the 13th ACM conference on computer and communications security, CCS 2006, Alexandria, VA, USA, October 30–November 3, 2006*, Association for Computing Machinery, 2006. See [11].

[46] Emilia Käsper, *Fast elliptic curve cryptography in OpenSSL*, in 2nd Workshop on Real-Life Cryptographic Protocols and Standardization (RLCPS 2011), to appear (2011). Citations in this document: §1, §1.

[47] Jonathan Katz, Nan Wang, *Efficiency improvements for signature schemes with tight security reductions*, in CCS 2003 [**8**] (2003), 155–164; portions incorporated into [**37**]. URL: http://www.cs.umd.edu/~jkatz/papers.html. Citations in this document: §2.

[48] Donald E. Knuth, *The art of computer programming, volume 3: sorting and searching*, 2nd edition, Addison-Wesley, Reading, 1998. ISBN 0-201-89685-0. Citations in this document: §5.

[49] Kaoru Kurosawa (editor), *Advances in cryptology — ASIACRYPT 2007, 13th international conference on the theory and application of cryptology and information security, Kuching, Malaysia, December 2–6, 2007, proceedings*, Lecture Notes in Computer Science, 4833, Springer, 2007. ISBN 978-3-540-76899-9. See [14].

[50] Chae Hoon Lim, Pil Joong Lee, *More flexible exponentiation with precomputation*, in [**29**] (1994), 95–107. Citations in this document: §4.

[51] Patrick Longa, Catherine H. Gebotys, *Efficient techniques for high-speed elliptic curve cryptography*, in CHES 2010 [**52**] (2010), 80–94. Citations in this document: §1, §1, §1.

[52] Stefan Mangard, François-Xavier Standaert (editors), *Cryptographic hardware and embedded systems, CHES 2010, 12th international workshop, Santa Barbara, CA, USA, August 17–20, 2010, proceedings*, Lecture Notes in Computer Science, 6225, Springer, 2010. ISBN 978-3-642-15030-2. See [51].

[53] Mitsuru Matsui (editor), *Advances in cryptology — ASIACRYPT 2009, 15th international conference on the theory and application of cryptology and information security, Tokyo, Japan, December 6–10, 2009, proceedings*, Lecture Notes in Computer Science, 5912, Springer, 2009. ISBN 978-3-642-10365-0. See [23].

[54] Alfred Menezes (editor), *Advances in cryptology — CRYPTO 2007, 27th annual international cryptology conference, Santa Barbara, CA, USA, August 19–23, 2007, proceedings*, Lecture Notes in Computer Science, 4622, Springer, 2007. ISBN 978-3-540-74142-8. See [30].

[55] David M'Raïhi, David Naccache, David Pointcheval, Serge Vaudenay, *Computational alternatives to random number generators*, in SAC '98 [**76**] (1999), 72–80. URL: http://www.di.ens.fr/~pointche/Documents/Papers/1998_sac.pdf. Citations in this document: §2.

[56] David Naccache (editor), *Topics in cryptology — CT-RSA 2001: the cryptographers' track at RSA Conference 2001, San Francisco, CA, USA, April 2001, proceedings*, Lecture Notes in Computer Science, 2020, Springer, 2001. ISBN 3-540-41898-9. MR 2003a:94039. See [22].

[57] David Naccache, David M'Raïhi, Françoise Levy-dit-Vehel, *Patent application WO/1998/051038: pseudo-random generator based on a hash coding function for cryptographic systems requiring random drawing* (1997). URL: http://www.wipo.int/pctdb/en/ia.jsp?IA=FR1998000901. Citations in this document: §2.

[58] David Naccache, David M'Raïhi, Serge Vaudenay, Dan Raphaeli, *Can D.S.A. be improved? Complexity trade-offs with the digital signature standard*, in Eurocrypt '94 [**28**] (1994). Citations in this document: §5, §5, §5, §5, §5, §5, §5.

[59] Michael Naehrig, Ruben Niederhagen, Peter Schwabe, *New software speed records for cryptographic pairings*, in Latincrypt 2010 [**5**] (2010), 109–123. URL: http://cryptojedi.org/users/peter/#dclxvi. Citations in this document: §3.

[60] Gregory Neven, Nigel P. Smart, Bogdan Warinschi, *Hash function requirements for Schnorr signatures*, Journal of Mathematical Cryptology **3** (2009), 69–87. URL: http://www.zurich.ibm.com/~nev/papers/schnorr.html. Citations in this document: §2, §2.

[61] Phong Q. Nguyen, Igor 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.

[62] Kaisa Nyberg (editor), *Advances in cryptology — EUROCRYPT '98, international conference on the theory and application of cryptographic techniques, Espoo, Finland, May 31–June 4, 1998, proceedings*, Lecture Notes in Computer Science, 1403, Springer, 1998. ISBN 3-540-64518-7. See [10].

[63] Josef Pieprzyk (editor), *Advances in cryptology — ASIACRYPT 2008, 14th international conference on the theory and application of cryptology and information security, Melbourne, Australia, December 7–11, 2008*, Lecture Notes in Computer Science, 5350, 2008. ISBN 978-3-540-89254-0. See [41].

[64] Nicholas Pippenger, *On the evaluation of powers and related problems (preliminary version)*, in FOCS '76 [**1**] (1976), 258–263; newer version split into [**65**] and [**66**]. MR 58:3682. URL: http://cr.yp.to/bib/entries.html#1976/pippenger. Citations in this document: §4, §5.

[65] Nicholas Pippenger, *The minimum number of edges in graphs with prescribed paths*, Mathematical Systems Theory **12** (1979), 325–346; see also older version [**64**]. ISSN 0025-5661. MR 81e:05079. URL: http://cr.yp.to/bib/entries.html#1976/pippenger.

[66] Nicholas Pippenger, *On the evaluation of powers and monomials*, SIAM Journal on Computing **9** (1980), 230–250; see also older version [**64**]. ISSN 0097-5397. MR 82c:10064. URL: http://cr.yp.to/bib/entries.html#1976/pippenger.

[67] David Pointcheval, Jacques Stern, *Security arguments for digital signatures and blind signatures*, Journal of Cryptology **13** (2000), 361–396. URL: ftp://ftp.di.ens.fr/pub/users/pointche/Papers/2000_joc.pdf. Citations in this document: §2.

[68] Bart Preneel (editor), *Progress in cryptology — AFRICACRYPT 2009, second international conference on cryptology in Africa, Gammarth, Tunisia, June 21–25, 2009, proceedings*, Lecture Notes in Computer Science, 5580, Springer, 2009. See [26].

[69] Bart Preneel, Stafford E. Tavares (editors), *Selected areas in cryptography, 12th international workshop, SAC 2005, Kingston, ON, Canada, August 11–12, 2005, revised selected papers*, Lecture Notes in Computer Science, 3897, Springer, 2006. ISBN 3-540-33108-5. MR 2007b:94002. See [7].

[70] Jothi Rangasamy, Douglas Stebila, Colin Boyd, Juan González Nieto, *An integrated approach to cryptographic mitigation of denial-of-service attacks*,

in ASIACCS 2011 [4] (2011). URL: http://www.douglas.stebila.ca/files/research/papers/RSBG11.pdf. Citations in this document: §1.

[71] Rainer A. Rueppel (editor), *Advances in cryptology — EUROCRYPT '92, workshop on the theory and application of cryptographic techniques, Balatonfüred, Hungary, May 24–28, 1992, proceedings*, Lecture Notes in Computer Science, 658, Springer, Berlin, 1993. ISBN 3-540-56413-6. MR 94e:94002. See [19].

[72] Claus P. Schnorr, *Efficient identification and signatures for smart cards*, in Crypto '89 [18] (1990), 239–252; see also newer version [73]. Citations in this document: §2, §2, §2.

[73] Claus P. Schnorr, *Efficient signature generation by smart cards*, Journal of Cryptology **4** (1991), 161–174; see also older version [72]. URL: http://www.mi.informatik.uni-frankfurt.de/research/papers.html.

[74] Claus P. Schnorr, Markus Jakobsson, *Security of discrete log cryptosystems in the random oracle + generic model* (2000). URL: http://www.mi.informatik.uni-frankfurt.de/research/papers.html. Citations in this document: §2.

[75] Jacques Stern, David Pointcheval, John Malone-Lee, Nigel P. Smart, *Flaws in applying proof methodologies to signature schemes*, in Crypto 2002 [80] (2002), 93–110. Citations in this document: §2.

[76] Stafford Tavares, Henk Meijer (editors), *Selected areas in cryptography, 5th annual international workshop, SAC98, Kingston, Ontario, Canada, August 17–18, 1998, proceedings*, Lecture Notes in Computer Science, 1556, Springer, 1999. ISBN 3-540-65894-7. See [55].

[77] Serge Vaudenay (editor), *Progress in cryptology — AFRICACRYPT 2008, First international conference on cryptology in Africa, Casablanca, Morocco, June 11–14, 2008, proceedings*, Lecture Notes in Computer Science, 5023, Springer, 2008. ISBN 978-3-540-68159-5. See [13].

[78] Ingo 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.

[79] John 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.

[80] Moti Yung (editor), *Advances in cryptology — CRYPTO 2002, 22nd annual international cryptology conference, Santa Barbara, California, USA, August 18–22, 2002, proceedings*, Lecture Notes in Computer Science, 2442, Springer, 2002. ISBN 3-540-44050-X. See [75].

[81] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *Public key cryptography — 9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, 2006. ISBN 978-3-540-33851-2. See [12].