Fast Elliptic Curve Cryptography in OpenSSL

Emilia Käsper[1,2]

[1] Google
[2] Katholieke Universiteit Leuven, ESAT/COSIC
emilia.kasper@esat.kuleuven.be

Abstract. We present a 64-bit optimized implementation of the NIST and SECG-standardized elliptic curve P-224. Our implementation is fully integrated into OpenSSL 1.0.1: full TLS handshakes using a 1024-bit RSA certificate and ephemeral Elliptic Curve Diffie-Hellman key exchange over P-224 now run at twice the speed of standard OpenSSL, while atomic elliptic curve operations are up to 4 times faster. In addition, our implementation is immune to timing attacks—most notably, we show how to do small table look-ups in a cache-timing resistant way, allowing us to use precomputation. To put our results in context, we also discuss the various security-performance trade-offs available to TLS applications.

Keywords: elliptic curve cryptography, OpenSSL, side-channel attacks, fast implementations

1 Introduction

1.1 Introduction to TLS

Transport Layer Security (TLS), the successor to Secure Socket Layer (SSL), is a protocol for securing network communications. In its most common use, it is the "S" (standing for "Secure") in HTTPS. Two of the most popular open-source cryptographic libraries implementing SSL and TLS are OpenSSL [19] and Mozilla Network Security Services (NSS) [17]: OpenSSL is found in, e.g., the Apache-SSL secure web server, while NSS is used by Mozilla Firefox and Chrome web browsers, amongst others.

TLS provides authentication between connecting parties, as well as encryption of all transmitted content. Thus, before any application data is transmitted, peers perform authentication and key exchange in a TLS *handshake*. Two common key exchange mechanisms in TLS are RSA key exchange and (authenticated) Diffie-Hellman (DH) key exchange. While RSA key exchange is computationally cheaper, DH key exchange provides the additional property of *perfect forward secrecy*. Our work was motivated from a practical viewpoint: after analyzing the overhead associated with forward secure cipher suites, we set out to improve the performance of Diffie-Hellman handshakes in OpenSSL. As a result, we describe a new optimized elliptic curve implementation that is integrated into OpenSSL and fully compatible with the elliptic curve flavour of DH handshakes in TLS.

1.2 Forward secrecy in TLS

In a typical TLS handshake, say, when the client is a browser connecting to an HTTPS server, authentication is unilateral, meaning that only the server is authenticated to the client. In an RSA handshake, authenticated key exchange is achieved via the following mechanism: the server sends its RSA public key together with a corresponding certificate; the client, upon successfully verifying the certificate, replies with the pre-master secret, encrypted with the

server's public key. Now, the server can decrypt the client's key exchange message and both parties derive a shared session key from the pre-master secret.

RSA handshakes exhibit a single point of failure: the security of all sessions relies on a single static key. If the server's private RSA key is compromised, the security of *all* sessions established under that key is violated. In other words, the attacker can record TLS traffic and decrypt these sessions later, should the key become compromised.

The complementary property, *forward secrecy*, which ensures that no long-term key compromise can affect the security of past sessions, is achieved in TLS via authenticated Diffie-Hellman (DH) handshakes. Contrary to RSA handshakes, the server's long-term RSA key now serves solely the purpose of authentication: it is only used to *sign* the server's DH value. If ephemeral DH is used, i.e., both parties generate a fresh DH value for every handshake, we achieve perfect forward secrecy, as the security of each session depends on a different instance of the DH problem.

While forward secrecy is undoubtedly a nice property to have, it comes at a cost. In an RSA handshake, the server needs to perform one private RSA operation (decryption); in a DH handshake, the server still needs a private RSA operation (signing) and, in addition, two exponentiations in the DH group. For more efficient Diffie-Hellman operations, TLS thus specifies an extension for elliptic curves, which achieve equivalent security with smaller group and field sizes (and hence, faster computation time).

Elliptic curve cryptography in TLS, as specified in RFC 4492 [7], includes elliptic curve Diffie-Hellman (ECDH) key exchange in two flavours: fixed-key key exchange with ECDH certificates; and ephemeral ECDH key exchange using an RSA or ECDSA certificate for authentication. While we focus our discussion on the ephemeral cipher suites providing perfect forward secrecy, our implementation results are also applicable to ECDH certificates and ECDSA signatures.

2 Motivation

2.1 Security parameter choices in TLS

An application choosing its TLS parameters should consider that the security of the session is bound not only by the security of the symmetric encryption algorithm, but also on the security of the key exchange algorithm used to establish the session key—a session using AES-128 still achieves only roughly 80-bit security if 1024-bit RSA key exchange is used. According to various key length recommendations [12, 18], in order to match 128-bit security, the server should use an RSA encryption key or a DH group of at least 3072 bits, or an elliptic curve over a 256-bit field, while a more feasible 2048-bit RSA key/DH group or a 224-bit elliptic curve still achieves 112 bits of security.

In settings where 2048-bit RSA is considered prohibitively slow, ECDH key exchange with a 1024-bit RSA signing key offers a neat security-performance trade-off—it is faster than plain 2048-bit RSA key exchange (see Sect. 4 for exact timings), while offering perfect forward secrecy.[3] Yet ECDH key exchange is still significantly slower than 1024-bit RSA key exchange. Currently, one 224-bit elliptic curve multiplication costs more in OpenSSL than a 1024-bit private RSA operation (and recall that the server needs *two* EC multiplications per handshake, while it needs only one RSA operation), so we focused our attention on optimizing

[3] While the 1024-bit RSA key still offers only 80 bits protection, its use as a *signing-only* key is less of a paradox, as the compromise of this key does not affect the security of past legitimate sessions.

the performance of the OpenSSL elliptic curve library. More specifically, as a lot of speed can be gained from implementing custom field arithmetic for a fixed field, we chose the NIST P-224 elliptic curve (`secp224r1` in [20]) as a target for our 64-bit optimized implementation.

2.2 Why NIST P-224?

Recently, several authors have published fast code for custom elliptic curves offering roughly 128 bits of security (see e.g. the SUPERCOP collected benchmarking results [11]). However, as our goal was to improve the performance of TLS handshakes in the OpenSSL library, we needed to ensure that the curve we choose is also supported by other common client libraries, and that the TLS protocol supports the negotiation of the curve.

Following the recommendations of the Standards for Efficient Cryptography Group [20], RFC 4492 specifies a list of 25 named curves for use in TLS, with field size ranging from 163 to 521 bits. Both OpenSSL and the Mozilla NSS library support all those curves. In addition, TLS allows peers to indicate support for unnamed prime and/or characteristic-2 curves (the OpenSSL elliptic curve library supports unnamed curves, while NSS does not). Yet the TLS specification has two important restrictions. First, it is assumed that the curve is of the form $y^2 = x^3 + ax + b$ (i.e., a Weierstrass curve), since the only parameters conveyed between the peers are the values a and b—many of the fastest elliptic curves today do not meet this format. Second, the client cannot indicate support for a specific unnamed curve in its protocol messages (that is, a client wishing to use unnamed curves must support all of them). Given these constraints, we chose to optimize one of the named curves, NIST P-224.[4]

Note that in order to provide 128-bit security, one of the two 256-bit named curves would have been a logical choice. Yet it happens that the 224-bit curve lends itself to a much faster implementation. Namely, an element of a 224-bit field fits easily in four 64-bit registers, while a 256-bit element needs five registers (it could fit in 4, but we also want to accommodate carry bits for efficient and timing-attack resistant modular reduction). An extra register, in turn, implies slower field arithmetic. For example, multiplication of two 5-register elements requires 25 64-bit multiplications, while multiplication of two 4-register elements requires only 16.

Aside from suitable field length, the NIST P-224 prime has a very simple format ($p = 2^{224} - 2^{96} + 1$) that further facilitates efficient field arithmetic. Indeed, Bernstein has already made use of this by implementing NIST P-224 for 32-bit platforms, using clever floating point arithmetic [2]. However, we chose to reimplement the curve from scratch, using more efficient 64-bit integer arithmetic, as well as adding side-channel protection.

2.3 Side-channel concerns

In addition to providing a fast implementation, we wanted to offer one that was constant-time and thus verifiably resistant to timing attacks. The lack of side-channel protection in the current OpenSSL elliptic curve library has already been successfully exploited by Brumley and Hakala [6] who mounted a cache-timing key recovery attack on the ECDSA portion of the library. While the same attack may not be feasible for ephemeral ECDH key exchange (assuming single-use keys), we feel it is prudent to ensure side-channel resistance for other possible applications, including ECDSA and ECDH-certificate-based key exchange.

[4] As noted in RFC 4492, curve monoculture can lead to focused attacks on a single curve; yet NIST P-224 offers a comfortable 112-bit security level.

3 NIST P-224 Implementation

Our 64-bit implementation of the NIST P-224 elliptic curve is written in C—the 128-bit data type available in GCC allows us to make use of the 64-bit registers, as well as the 64-bit unsigned integer multiplication instruction MUL, which stores the 128-bit result in two 64-bit registers.

Our implementation does not rely on platform-specific instruction set extensions such as SSE. Of the SSE instructions, the one potentially useful to us is the packed multiplication PMULUDQ, which can do two unsigned 32-bit-to-64-bit integer multiplications in one cycle. While PMULUDQ is beneficial for Intel processors, MUL is best on AMDs [4]—we target both platforms, so opted to use the latter, as using 64-bit limbs also makes modular reduction simpler.

3.1 Field arithmetic

We represent elements of the 224-bit field as polynomials $a_0 + 2^{56}a_1 + 2^{112}a_2 + 2^{168}a_3$, where each coefficient a_i is an unsigned 64-bit integer. (Notice that a field element can have multiple such representations—we only reduce to the unique minimal representation at the end of the computation.) Outputs from multiplications are represented as unreduced polynomials $b_0+2^{56}b_1+2^{112}b_2+2^{168}b_3+2^{224}b_4+2^{280}b_5+2^{336}b_6$, where each b_i is an unsigned 128-bit integer. Using this representation, field multiplication costs 16 64-bit-to-128-bit multiplications and 9 128-bit additions, while squaring costs 10 multiplications, 6 scalar multiplications by 2, and 3 additions.

Aside from multiplications, we also need linear operations. Scalar multiplication and addition are straightforward. To perform subtraction $a - b$, we add a suitable multiple of the field prime (i.e., a "multiple" of zero) to the left operand a, ensuring that $a > b$—we can then perform unsigned subtraction.

Between two subsequent multiplications, we reduce the coefficients partially, ensuring that the four output coefficients satisfy $a_i < 2^{57}$. For each field operation, we also assert input bounds to guarantee that the output does not overflow. For example, we need to ensure that all input coefficients to an addition satisfy $a_i < 2^{63}$, in order to guarantee that the output coefficients satisfy $b_i < 2^{63} + 2^{63} = 2^{64}$ and thus, fit in a 64-bit unsigned integer without overflow.

3.2 Elliptic curve point operations

For elliptic curve group operations, we use the well-known formulae in Jacobian projective coordinates: point doubling in projective coordinates costs 5 field squarings, 3 field multiplication, and 12 linear operations (additions, subtractions, scalar multiplications), while point addition costs 4 squarings, 12 multiplications and 7 linear operations.

In order to minimize computation cost, we have manually analyzed the computation chain of point addition and doubling. By bounding inputs to each step, we perform modular reductions if and only if the next operation could overflow. For example, starting with partially reduced inputs x and y, we can compute $3(x + y)(x - y)$ without intermediate reductions. When computing $3(x + y)$, the coefficients of the output satisfy $a_i < 3 \cdot (2^{57} + 2^{57}) < 2^{60}$. The largest scalar added to a coefficient of the left operand of a subtraction is $2^{58} + 2$, so the coefficients of $x - y$ satisfy $a_i < 2^{57} + 2^{58} + 2 < 2^{59}$. Finally, as we use 4 limbs, each coefficient

of the product is the sum of at most 4 atomic products: $b_i < 4 \cdot 2^{60} \cdot 2^{59} = 2^{121}$. The result fits comfortably in 128 bits without an overflow. We computed these bounds for each execution step: overall, the whole computation only needs 15 reductions for point addition and 7 for point doubling.

Finally, as elliptic curve points in TLS are transmitted using affine coordinates, we need a conversion routine from Jacobian to affine coordinates. As a Jacobian point (X, Y, Z) corresponds to the affine point $(X/Z^2, Y/Z^3)$, this conversion requires a field inversion—computing $Z^{-1} = Z^{p-2} \bmod p$ can be done in 223 field squarings, 11 field multiplications and 234 modular reductions [2].

3.3 Point multiplication with precomputation

Binary (schoolbook) elliptic curve point multiplication of nP requires 224 point doublings and on average 112 point additions for a 224-bit scalar n. In order to reduce the cost, we use standard precomputation techniques. By computing 16 multiples of the point P—$0 \cdot P, 1 \cdot P, \ldots, 15 \cdot P$—in 7 doublings and 7 additions, we bring the point multiplication cost down to 224 doublings and 56 additions (a total of 231 doublings and 63 additions, including precomputation).

For a fixed point G, we can perform interleaved multiplication by precomputing 16 linear combinations of the form $b_0G + b_1G^{56} + b_2G^{112} + b_3G^{168}$, where $b_i \in \{0, 1\}$.[5] As well as including precomputed multiples for the NIST standard generator G, our implementation allows the application to perform this precomputation for a custom generator. After precomputation, each subsequent multiplication with the generator costs 56 doublings and 56 additions.

Finally, our implementation also supports batch multiplication. Namely, we amortize the cost of doublings by computing a linear combination of k points $n_1P_1 + \cdots + n_kP_k$ in an interleaved manner [16]: the full computation still costs $56k$ additions, but only 224 (rather than $224k$) doublings. This technique is immediately useful in ECDSA signature verification.

3.4 Side-channel protection

Kocher [14] was the first to show that the execution time of a cryptographic algorithm may leak significant information about its secrets. In software implementations, two important leakage points have been identified: (i) conditional branching dependent on the secret input; and (ii) table lookups using secret-dependent lookup indices. Branching leaks information if the branches require different execution time, but worse, the branch prediction used in modern CPUs causes a timing variance even for equivalent branches [1]. Table lookups are vulnerable as lookup results are stored in processor cache: simply put, multiple lookups into the same table entry are faster than multiple lookups into different locations, as the results are fetched from cache rather than main memory. Thus, the best way to ensure side-channel resistance is to avoid branching and table lookups altogether.

Our implementation is constant-time for single point multiplication. To ensure that it does not leak any timing information about the secret scalar, we have used the following techniques:

– Field arithmetic is implemented using 64-bit arithmetic and Boolean operations only—there are no conditional carries and no other branches;

[5] As this precomputation is almost as expensive as a full point multiplication, it is only useful when the point G is used more than once.

Listing 1 A routine for choosing between two inputs `a` and `b` in constant time, depending on the selection bit `bit`

```
int select (int a, int b, int bit) {
    /* -0 = 0, -1 = 0xff....ff */
    int mask = - bit;
    int ret = mask & (a^b);
    ret = ret ^ a;
    return ret;
}
```

Listing 2 A cache-timing resistant table lookup.

```
int do_lookup(int a[16], int bit[4]) {
    int tmp0[8], tmp1[4], tmp2[2];
    /* select values where the least significant bit of the index is bit[0] */
    tmp0[0] = select(a[0], a[1], bit[0]); tmp0[1] = select(a[2], a[3], bit[0]);
    tmp0[2] = select(a[4], a[5], bit[0]); tmp0[3] = select(a[6], a[7], bit[0]);
    tmp0[4] = select(a[8], a[9], bit[0]); tmp0[5] = select(a[10], a[11], bit[0]);
    tmp0[6] = select(a[12], a[13], bit[0]); tmp0[7] = select(a[14], a[15], bit[0]);
    /* select values where the second bit of the index is bit[1] */
    tmp1[0] = select(tmp[0], tmp[1], bit[1]); tmp1[1] = select(tmp[2], tmp[3], bit[1]);
    tmp1[2] = select(tmp[4], tmp[5], bit[1]); tmp1[3] = select(tmp[6], tmp[7], bit[1]);
    /* select values where the third bit of the index is bit[2] */
    tmp2[0] = select(tmp2[0], tmp[1], bit[2]); tmp2[1] = select(tmp2[2], tmp2[3], bit[2]);
    /* select the value where the most significant bit of the index is bit[3] */
    ret = select(tmp3[0], tmp3[1], bit[3]);
    return ret;
}
```

– Rather than skipping unnecessary operations, point multiplication performs a dummy operation with the point-at-infinity whenever necessary (for example, leading zeroes of the scalar are absorbed without leaking timing information);
– Secret-dependent lookups into the precomputation table are performed in constant time, ensuring that no cache-timing information leaks about the secret scalar.

While branch-free field arithmetic and constant-time multiplication algorithms are also seen in some other implementations (e.g., in the constant-time implementations of Curve25519 [3, 15]), combining secret-dependent lookups into the precomputation table with side-channel resistance is more tricky. Joye and Tunstall suggest to secure modular exponentiations by adding a random multiple of the group order to the secret exponent [13]. Using a different mask at every execution limits the leak, as multiple measurements on the same secret cannot be easily linked. The same technique could be employed for safeguarding the secret scalar in point multiplication, however, the masked scalar has a longer bit representation, thus requiring more operations—the overhead is worse for elliptic curves, which have shorter exponents compared to, say, RSA. Instead, we devised a solution for performing lookups in a way that leaks no information even from a single execution.

Listing 1 shows sample code for implementing an **if**-statement in constant time: the routine `select()` returns input `a` if the input bit `bit` equals 0, and returns `b` if `bit` equals 1. By repeating `select()` 15 times on point inputs, we can thus select the correct precomputed point in a secure manner (see Listing 2)—independent of the lookup index $(\mathsf{bit}[3], \mathsf{bit}[2], \mathsf{bit}[1], \mathsf{bit}[0])$, we loop through the whole precomputation table in a fixed order. While the execution time is still dependent on cache behaviour, the timing variance is *independent* of the secret lookup

index, thus leaking no valuable timing information. This strategy obviously doesn't scale for large tables, yet for us it is cheap compared to the cost of elliptic curve operations—we save more by precomputation than we lose by adding side-channel protection to the lookups.

4 Performance Results

4.1 The benchmark set-up

As our goal was to provide a fully integrated implementation for applications using OpenSSL, we also chose to measure performance directly within OpenSSL. Rather than counting cycles for stand-alone point multiplication, the OpenSSL toolkit allows us to report timings for complete OpenSSL operations, from a single ECDH comutation to a complete TLS handshake. As these results take into account any overhead introduced by the library, they depict the actual performance gain when switching from the standard implementation to our optimized version. At the same time, they can be viewed as an upper bound to atomic elliptic curve operations.

Our benchmark machine was `ambrel`, with the following parameters:

`ambrel`			
CPU	Intel Core 2 Duo E8400	Lithography	45nm
CPU frequency	3.0 GHz	RAM	4GB
OS	Linux 2.6.18-194.11.4.el5 x86_64	Compiler	gcc 4.4.4

Table 1. Our benchmark machine

All benchmarks were performed, utilizing a single core.

4.2 Results

For atomic operations, we benchmarked ECDH key generation as well as shared secret computation. The former case corresponds to multiplication with a fixed basepoint; the latter amounts to one multiplication with a random basepoint, plus input point validation and output point conversion to its affine representation. Our benchmarks for the shared secret computation include all these operations. In order to complete an ephemeral ECDH handshake with single-use keys, both parties need to compute one operation of each type.

In addition, we measured throughput for ECDSA signature generation and verification: ECDSA signing requires one fixed-point multiplication, while ECDSA verification makes use of batch multiplication to compute a linear combination of one fixed and one random point.

For benchmarking TLS handshakes, we chose the most common configuration: one-sided (server only) authentication using RSA certificates. The Qualys Internet SSL Survey from July 2010 [21] reveals that virtually all trusted SSL certificates contain either a 1024-bit or 2048-bit RSA key, so we restricted our attention to those key sizes. Since performance is crucial on the server side (and a typical browser client does not use the OpenSSL library anyway), we measured the time it takes a server to complete the handshake. Note that these timings include only computation time and do *not* reflect the communication latency. However, ephemeral ECDH key exchange or, more precisely, the requirement to maintain perfect forward secrecy can have an effect on handshake latency for typical TLS configurations—we

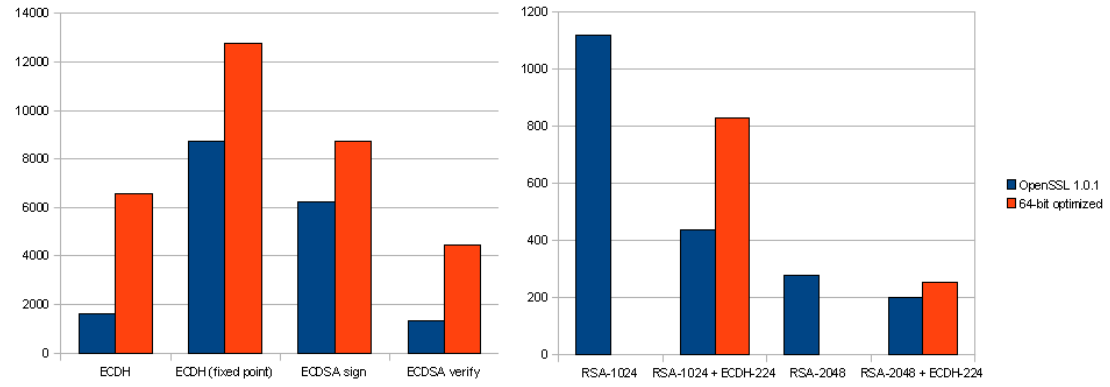

Fig. 1. Improved throughput (in operations/second) for elliptic curve operations (left) and TLS handshakes (right), measured on `ambrel`

OpenSSL 1.0.1	**ECDH (shared secret)**	**ECDH (keygen)**	**ECDSA sign**	**ECDSA verify**
standard	1602.9	8757.1	6221.0	1309.9
64-bit opt	6552.9	12789.2	8757.6	4442.9
OpenSSL 1.0.1	**RSA-1024**	**RSA-1024, ECDH-224**	**RSA-2048**	**RSA-2048, ECDH-224**
standard	1118.6	435.5	277.6	199.4
64-bit opt	—	826.4	—	253.4

Table 2. Throughput of NIST P-224 elliptic curve computations and TLS handshakes in operations/second, measured on `ambrel`

discuss some latency reduction mechanisms in TLS, and their compatibility with forward secrecy in Section 5.

Figure 1 illustrates the increased throughput (in operations/second) when switching from standard OpenSSL to our optimized implementation; the corresponding precise measurements are given in Table 2. Since the OpenSSL library already contains optimized code for fixed-point multiplication, the gain is highest for random point multiplication, where throughput increases from 1600 to over 6500 operations/second. Also, Diffie-Hellman handshakes with a 1024-bit RSA signing key are nearly twice as fast when using the optimized code (435 vs 826 handshakes/second), allowing to switch from plain RSA to forward-secure cipher suites with only a 26% drop in server throughput.

4.3 Comparison with other results

We benchmarked Bernstein's implementation of NIST P-224 on `ambrel`, using the timing software provided with the code—raw point multiplication is about 1.5 times slower than our fully integrated OpenSSL implementation. Brown et. al. also report timings of several NIST curves [5]. This software appears to be much slower than Bernstein's but the measurements are obtained on a Pentium II, so we omit exact cycle counts to avoid comparing apples to oranges.

For the curious reader, we have also gathered some benchmarking data for other elliptic curves in Table 3. We benchmarked `curve25519-donna`, a 64-bit constant-time implementation of Curve25519 [15] on `ambrel`, noting that our NIST P-224 implementation outperforms it despite any OpenSSL overhead (admittedly, NIST P-224 also offers a slightly lower security level compared to Curve25519). For comparison, we also give some figures for the

curve	impl.	platform	benchmarking suite	key gen.	shared secret comp.	security	const-time
NIST P-224	this paper	`ambrel`	OpenSSL	234573	457813	112 bits	yes
NIST P-224	Bernstein	`ambrel`	Bernstein	662220	662220	112 bits	no
curve25519	donna	`ambrel`	donna	≈540000	≈ 540000	≈128 bits	yes
curve25519	mpfq	`boing`	SUPERCOP	394254	381375	≈128 bits	no(?)
gls1271	eBATS	`boing`	SUPERCOP	140355	314730	≈128 bits	no(?)

Table 3. Selected benchmarking results for various curves, reported in cycles/operation

fastest Curve25519 implementation, as well as for the Galbraith-Lin-Scott implementation of a twisted Edwards curve over a field with $(2^{127} - 1)^2$ elements [9], as reported by the ECRYPT benchmarking suite SUPERCOP [11]. These implementations are, as far as we know, not constant-time.

From SUPERCOP, which reports performance on a variety of platforms, we chose the figures obtained from `boing`, a machine with a CPU identical to `ambrel`. Nevertheless, we stress that these are timings obtained via different benchmarking tools, on different machines, and as such, are only meant to give a very rough context to our results.

Finally, Bernstein et. al. also report extremely fast software for 192-bit modular arithmetic on 64-bit platforms [4]. Utilizing parallelism from hyperthreading on all 4 cores on an Intel Core 2 Quad, they are able to carry out over 100 million modular multiplications per second. Within our elliptic curve computation, we do about 10 million 224-bit modular multiplications, and another 10 million modular squarings per second on a single core—but due to the completely different setting, these results are not directly comparable.

5 Security considerations

5.1 Ephemeral versus fixed keys

For (Elliptic Curve) Diffie-Hellman key exchange, TLS does not strictly mandate the use of ephemeral keys—in order to save computation, the server may reuse its Diffie-Hellman value for multiple connections. On one hand, we note that our implementation is resistant to timing-attacks and thus, we deem it safe to reuse the secret Diffie-Hellman value. On the other hand, as our implementation includes an optimization for fixed basepoints, computing a new Diffie-Hellman value is by far the cheapest of the three public key operations required in an ECDH-RSA handshake and thus, by reusing DH secrets, the application potentially stands to lose more security than it stands to gain in performance.

5.2 Latency reduction mechanisms and forward secrecy

A full SSL/TLS handshake takes two round trips between the server and the client. SSL/TLS includes several mechanisms for reducing this latency:

- **Session caching** keeps session information (including the session key) in server-side cache; clients can resume previous sessions by presenting the corresponding session ID.
- **TLS session tickets** [8] allow stateless session resumption: the session information is now sent to the client in a session ticket encrypted with the server's long-term key.
- **False Start** allows "optimistic" clients to start sending (encrypted) application data before the handshake is finished.

All these mechanisms cut the handshake latency down to one round trip. Yet care should be taken when using them in conjunction with DH ciphers. Both session caching and tickets conflict with perfect forward secrecy. In particular, session tickets invalidate any forward secrecy completely, as an adversary having control over the server's long-term private key can decrypt the ticket to obtain the session key. To facilitate forward secrecy, latest versions of OpenSSL allow to selectively disable session caching and tickets for forward-secure cipher suites.

In contrast, False Start is perfectly compatible with forward-secure ciphers. In fact, the False Start Internet-Draft [10] recommends that clients should *only* false start with forward-secure cipher suites, in order to avoid cipher suite downgrade attacks by rogue servers. Thus, we conclude that it is possible to maintain perfect forward secrecy without sacrificing communication latency.

The Source Code

This software will be released in OpenSSL 1.0.1, and is available in the latest snapshots at ftp://ftp.openssl.org/snapshot/. Please refer to the release notes to compile and test the implementation.

Acknowledgements

The author is grateful to Bodo Möller and Adam Langley for their comments on the OpenSSL implementation.

References

1. Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology - CT-RSA 2007, The Cryptographers' Track at the RSA Conference 2007, San Francisco, CA, USA, February 5-9, 2007, Proceedings*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242. Springer, 2007.
2. Daniel J. Bernstein. A software implementation of NIST P-224, 2001. `http://cr.yp.to/nistp224.html`.
3. Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.
4. Daniel J. Bernstein, Hsueh-Chung Chen, Ming-Shing Chen, Chen-Mou Cheng, Chun-Hung Hsiao, Tanja Lange, Zong-Cing Lin, and Bo-Yin Yang. The billion-mulmod-per-second pc. In *Workshop record of SHARCS'09: Special-purpose Hardware for Attacking Cryptographic Systems*, 2009.
5. Michael Brown, Darrel Hankerson, Julio López, and Alfred Menezes. Software implementation of the nist elliptic curves over prime fields. In *Topics in Cryptology - CT-RSA 2001, The Cryptographer's Track at RSA Conference 2001, San Francisco, CA, USA, April 8-12, 2001, Proceedings*, volume 2020 of *Lecture Notes in Computer Science*, pages 250–265. Springer, 2001.
6. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In 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*, volume 5912 of *Lecture Notes in Computer Science*, pages 667–684. Springer, 2009.
7. Internet Engineering Task Force. Elliptic curve cryptography (ECC) cipher suites for transport layer security (TLS), 2006. `http://www.ietf.org/rfc/rfc4492`.
8. Internet Engineering Task Force. Transport layer security (TLS) session resumption without server-side state, 2008. `http://www.ietf.org/rfc/rfc5077`.
9. Steven D. Galbraith, Xibin Lin, and Michael Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 518–535. Springer, 2009.

10. TLS Working Group. Transport layer security (TLS) false start. https://tools.ietf.org/html/draft-bmoeller-tls-falsestart-00.
11. ECRYPT II. eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to/supercop.html.
12. ECRYPT II. Yearly report on algorithms and keysizes (2010), D.SPA.13 Rev. 1.0, ICT-2007-216676, 2010. http://www.ecrypt.eu.org/documents/D.SPA.13.pdf.
13. Marc Joye and Michael Tunstall. Exponent recoding and regular exponentiation algorithms. In Bart Preneel, editor, *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, volume 5580 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2009.
14. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO 96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.
15. Adam Langley. curve25519-donna: A 64-bit implementation of Curve25519. http://code.google.com/p/curve25519-donna/.
16. Bodo Möller. Algorithms for multi-exponentiation. In *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 165–180. Springer, 2001.
17. mozilla.org. Network Security Services. http://www.mozilla.org/projects/security/pki/nss/.
18. NIST. Recommendation for key management, special publication 800-57 part 1.
19. The OpenSSL project. OpenSSL—cryptography and SSL/TLS toolkit. http://www.openssl.org.
20. Certicom Research. SEC 2: Recommended elliptic curve domain parameters, 2010.
21. Ivan Ristic. Internet SSL survey 2010. Technical report, Qualys, 2010. Black Hat USA 2010.