Practical Invalid Curve Attacks on TLS-ECDH

Tibor Jager(✉), Jörg Schwenk(✉), and Juraj Somorovsky(✉)

Horst Görtz Institute for IT Security, Ruhr University Bochum,
Bochum, Germany
tibor.jager@rub.de

Abstract. Elliptic Curve Cryptography (ECC) is based on cyclic groups, where group elements are represented as points in a finite plane. All ECC cryptosystems implicitly assume that only valid group elements will be processed by the different cryptographic algorithms. It is well-known that a check for group membership of given points in the plane should be performed before processing.

However, in several widely used cryptographic libraries we analyzed, this check was missing, in particular in the popular ECC implementations of Oracle and Bouncy Castle. We analyze the effect of this missing check on Oracle's default Java TLS implementation (JSSE with a SunEC provider) and TLS servers using the Bouncy Castle library. It turns out that the effect on the security of TLS-ECDH is devastating. We describe an attack that allows to extract the long-term private key from a TLS server that uses such a vulnerable library. This allows an attacker to impersonate the legitimate server to any communication partner, after performing the attack only once.

1 Introduction

Elliptic Curve Cryptography (ECC) is one of the cornerstones of modern cryptography, due to its security and performance features. It is implemented in nearly every cryptographic application, ranging from Bluetooth device level encryption to securing cloud applications via TLS. Mathematically speaking, an elliptic curve is a set of points in a plane (in cryptography: a finite plane), together with a single (associative) operation, namely *point addition*. The set of points are those that satisfy an equation of the form

$$y^2 = x^3 + ax + b \tag{1}$$

and point addition can be defined as a geometric operation in the plane. Each set of elements together with an operation forms an algebraic *group*, if the set is closed under the given operation, if the operation is associative, and if a neutral element exists. Elliptic curves satisfy all these axioms, and thus can be used in any cryptosystem that operates on a mathematical group. For cryptographic applications it is also required that certain assumptions hold in the group G, e.g., the hardness of the discrete logarithm problem, or the CDH and DDH assumptions. An elliptic curve $E_{a,b}$ therefore has to be chosen carefully to guarantee that these assumptions hold.

G. Pernul et al. (Eds.): ESORICS 2015, Part I, LNCS 9326, pp. 407–425, 2015.
DOI: 10.1007/978-3-319-24174-6_21

Now a finite plane also contains points *outside* the elliptic curve $E_{a,b}$, and thus these points are not group elements of G. However they resemble group elements: They have two coordinates, and the functions defining the group operation can be applied to them. They just don't satisfy Eq. (1) with the given parameters a and b. If we use these points with the functions defining our EC cryptosystem, we may get strange results, since the group laws may not apply to them, or they may lie in a diffrent group where the cryptographic assumptions are not valid.

So strictly speaking, any cryptographic application using a cyclic group G should check that any operand that is supposed to be a group element of G is indeed contained in G. Indeed, it is well-known that this check is *in general* necessary to provide security [2,3,6,15]. *But is this check always implemented in the cryptographic libraries that are used in practical applications?*

It is also not clear for which *specific* applications this is inherently necessary. Even though it is considered good practice to *always* perform a test of group membership, we show that sometimes developers of even popular implementations of elliptic curve cryptography *omit* the check. *Which impact does a missing check of group membership have on the* specific *application TLS?*

To answer these questions, we studied the eight most important cryptographic libraries, which are used in TLS-ECDH (and *many* other applications). We found that a check for group membership was missing in three of these libraries, and this omitted check allows to compromise the security of a TLS implementation completely in two libraries (Oracle SunEC, Bouncy Castle), provided that a TLS-ECDH cipher suite is used.

TLS-(EC)DH. Transport Layer Security (TLS) is a security standard originally designed to protect HTTP traffic, but which is today used as a de facto security standard for many applications, e.g. EAP-(T)TLS, IMAPS and secure websockets. TLS consists of two main parts: The *Record Layer encryption*, which protects transported data using a MAC-then-PAD-then-ENCRYPT approach, and the *Handshake Protocol*, which negotiates cryptogra pgorihms and keys to be used by the Record Layer. Three different types of key agreement can be used in the Handshake protocol:

- TLS-RSA: The client chooses a random `PreMasterSecret` *pms*, encrypts it with the RSA public key of the server (contained in the server certificate), and sends this cryptogram to the server.
- TLS-DH: Here the server certificate contains a static Diffie-Hellman share g^s, and the client chooses a fresh DH share g^x. The `PreMasterSecret` is computed as $pms := (g^s)^x = (g^x)^s$.
- TLS-DHE: Here the server also chooses a fresh DH share g^y, and signs this value (plus some additional values). This signature can be verified using the server certificate. The `PreMasterSecret` is computed as $pms := (g^y)^x = (g^x)^y$.

Since only a mathematical group structure is required in the Diffie-Hellman key exchange, we can also use elliptic curves in the last two key agreement

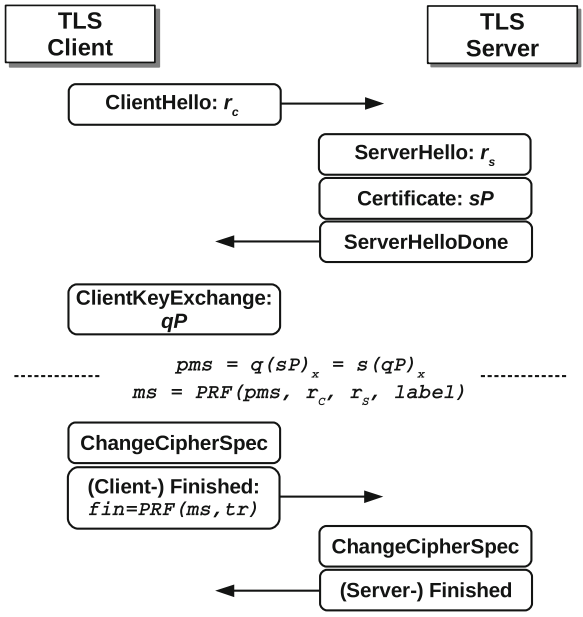

Fig. 1. Structure of the SSL/TLS Handshake protocol for `TLS_ECDH` cipher suites.

schemes. These variants are denoted as TLS-ECDH and TLS-ECDHE, respectively. The attacks described in this paper are applicable to TLS-ECDH. The structure of this handshake is described in Fig. 1. Our goal is to compute the private server key s. We may learn the public server key sP from the server certificate sent in the `Certificate` message, but since the Discrete Logarithm assumption (DLP)[1] holds in the elliptic curve group, we cannot compute s from this value.

Attacks on TLS. TLS can be attacked at three points: At the TLS handshake, at the Record Layer, and by using a specific TLS extension. The impact of each attack may range from low to high criticality.

Except where weak export cipher suites were used, the TLS Record Layer seemed secure. This situation changed with the BEAST attack published in 2011 [19]. Although the impact of this attack was low, it showed the practical vulnerability of the MAC-then-PAD-then-ENCRYPT scheme used. Critical attacks followed soon: Lucky 13 [1] and POODLE [17]. However, with these attacks only parts of the plaintext exchanged could be decrypted, and thus the criticality lay in the fact that e.g. HTTP session cookies could be decrypted.

[1] Group operations can be written as additive or multiplicative operations. Elliptic curves traditionally use additive notation, so for EC this assumption could be relabeled "discrete factor assumption". However, DLP is the standard term used for this assumption.

The first critical attack on the TLS handshake, which is hard to mitigate and thus resurfaces from time to time, is the famous adaptive chosen ciphertext attack by Daniel Bleichenbacher [5]. With this attack, a single TLS session could be completely broken by computing the `PreMasterSecret` from an intercepted `ClientKeyExchange` message, and from server error messages or timing measurements. Another example of an attack on the TLS handshake is the attack by Brumley et al., who analyzed a bug in EC computation of OpenSSL [6]. The bug allowed the authors to apply practical attacks against TLS servers using NIST secp256r1 and secp384r1 curves, and to extract EC private keys.

Even more critical was the Heartbleed vulnerability,[2] which was not based on a cryptographic attack, but on an implementation error of the OpenSSL Heartbeat extension: An attacker could read the server's private key directly from the memory of the OpenSSL process.

In this paper we describe a cryptographic attack on the TLS handshake which also recovers the private key of the server. Our attack is however less critical than Heartbleed, most importantly because the widely-used OpenSSL library is not affected, and TLS-DH cipher suites are less frequently used in practice than TLS-DHE or TLS-RSA cipher suites.

Our Attack. As a starting point, we used the invalid curve attack sketched by Biehl et al. in [3] and explained in more detail by Antipa *et al.* [2]. The basic idea is to define several different elliptic curves in the same plane as the original curve, by varying the parameter b. The groups defined by these curves may have arbitrary order within a certain range, and this order may be divisible by small primes $2, 3, 5, 7, 11, ...$

For example, if we find a parameter b' where the order of the corresponding group is divisible by 7, then we can find a point P' on this curve that generates a subgroup of order 7. If we send this point P' to the TLS server, then there are only 7 different values for sP'. Thus if we could learn sP', we could compute s mod 7. If we do this for enough different small primes, we can apply the Chinese Remainder theorem to compute the private server key s.

This attack however only works if the result of the EC computation is directly available to the adversary, which is not the case for TLS-ECDH: The resulting value sP' is only used internally by the server as the `PreMasterSecret` *pms*. *Thus we never directly see this value, but we can guess this value and check it against the server.*

Therefore we used the strategy of Brumley et al. [6], and adapted the attack on TLS-ECDH in the following way:

1. We start like in the attacks of [2,3] by generating several different curves with subgroups of small prime order.
2. For each of these small prime orders p_i, we send a generator G_i of the corresponding subgroup in the `ClientKeyExchange` message to the server.

[2] http://heartbleed.com/.

3. Additionally, we guess the value sG_i, which can only be one of the p values generated by G_i. Using this guessed value as the `PreMasterSecret` pms, we compute the `MasterSecret` ms and the `ClientFinished` message.
4. If we guessed correctly, the server will accept the `ClientFinished` message, and respond with the `ServerFinished` message. In that case, we have learned $s^2 \mod p_i$.

Results. We studied eight TLS-ECDH implementations. TLS servers based on Oracle's default Java TLS implementation using the SunEC provider, and the Bouncy Castle library were vulnerable to the presented attack. The WolfSSL library did not validate EC points, but it was not vulnerable. We provide an explanation for this behavior in Appendix A. The results are summarized in Table 1.

Table 1. Overview on the tested libraries

Lib	Bouncy Castle Java 1.50	MatrixSSL 1.3.10	mbed TLS 1.3.10	OpenSSL 1.0.2a	LibreSSL 2.1.6	SunEC Security Provider 1.8	SunPKCS11-NSS Security Provider 1.7	WolfSSL 3.4.6
point check?	no	yes	yes	yes	yes	no	yes	no
vuln.?	yes	no	no	no	no	yes	no	no

We were able to perform the attack against a TLS server with a SunEC provider with about 3300 server queries, and a server based on the Bouncy Castle library with about 17,000 server queries. Both test servers used the secp256r1 NIST curve. The significantly larger values for SunEC resulted from an unidentified computation error in the ECC library:[3] Certain computations resulted in False Positives, and the probability for False Positives was proportional to the inverse of the size of the chosen small group. Thus we had to choose larger primes for our attack, and consequently the average number of guesses increased.

Contribution. The contributions of this paper are the following:

- We adapt attacks of [2,3] to TLS-ECDH, and present a representative study on TLS libraries using TLS-ECDH cipher suites.
- We show that three out of eight analyzed libraries do not include curve point validations, and that two of them are vulnerable to invalid curve attacks. This allowed us to reveal TLS long-term private keys with a few thousands of server requests.
- We present a modified algorithm that allowed us to attack a TLS server using the SunEC security provider even in the presence of invalid EC computations, with high probability.

[3] We were not able to investigate this in more detail, because the source code is not publicly available.

– We give additional practical arguments why group membership checks are of prime importance in cryptographic applications.

2 Invalid Curve Attacks on ECC

2.1 A Brief Recap of Elliptic Curve Cryptography

In this section we give a brief introduction to elliptic curve cryptography, mainly in order to introduce our notation. We refer to [7,13] for a more verbose treatment of elliptic curves.

Let $\mathbb{F}$ be a finite field (e.g., $\mathbb{F} = \mathbb{Z}_p$ for prime p) with characteristic not equal to 2 or 3. An elliptic curve in Weierstrass form over $\mathbb{F}$ is described by *curve parameters* $\pi := (\mathbb{F}, a, b)$, where $a, b \in \mathbb{F}$. Let

$$E_\pi := \{(x, y) \in \mathbb{F}^2 : y^2 = x^3 + ax + b\} \cup \{O_\infty\}$$

denote the set of solutions (x, y) to the Weierstrass equation $y^2 = x^3 + ax + b$ over $\mathbb{F}$ defined by π, along with a special symbol O_∞ which is called the *point at infinity*. Let $+_\pi : E_\pi \times E_\pi \to E_\pi$ denote the map that takes as input two points $P, Q \in \mathbb{F}^2$ and outputs the point $R \in \mathbb{F}^2$ computed as

$$R = P +_\pi Q := \begin{cases} \texttt{ADD}_\pi(P, Q) & \text{if } P \neq Q, \\ \texttt{DBL}_\pi(P) & \text{if } P = Q. \end{cases}$$

Here, $\texttt{ADD}_\pi$ and $\texttt{DBL}_\pi$ denote the algorithms depicted in Fig. 2.

$\texttt{ADD}(P, Q)$:	$\texttt{DBL}(P)$:
$(x_P, y_P) := P; (x_Q, y_Q) := Q$	$(x_P, y_P) := P$
If $P = O_\infty$ **then Return** Q	**If** $P = O_\infty$ **then Return** P
If $Q = O_\infty$ **then Return** P	$\lambda := (3x_p^2 - a)/(2y_P)$
$\lambda := (y_P - y_Q)/(x_P - x_Q)$	$x_R := \lambda^2 - 2x_P$
$x_R := \lambda^2 - x_P - x_Q$	$y_R := y_P + \lambda(x_R - x_P)$
$y_R := y_P + \lambda(x_R - x_P)$	**Return** (x_R, y_R)
Return (x_R, y_R)	

Fig. 2. Algorithms DBL and ADD for point doubling and addition. Note that both algorithms are independent of the curve parameter b.

Remark 1. Note that algorithm $\texttt{ADD}_\pi$ depends only on P, Q, and the field $\mathbb{F}$, but not on the curve parameters a and b. Similarly, $\texttt{DBL}_\pi$ depends only on P, the field $\mathbb{F}$ and curve parameter a, but not on curve parameter b. Thus, *the computation of the group operation* $+_\pi$ *is independent of curve parameter* b. This is a crucial property for the attack described below.

The set of points E_π along with the group law $+_\pi$ forms an algebraic group $\mathbb{G}_\pi = (E_\pi, +_\pi)$. We will write $P + Q$ shorthand for $P +_\pi Q$ when the reference to parameters π is clear. For $n \in \mathbb{N}$ we write nP for the n-fold sum $P + \cdots + P$.

In the sequel we will furthermore write $[P]_x$ to denote the x-coordinate of a point P. If P is the point at infinity, we set $[P]_x := \emptyset$, where $\emptyset$ is an arbitrary constant.

2.2 Invalid Curve Attacks on Elliptic Curves in TLS

The idea of small subgroup attacks is due to Lim and Lee [14], who described such attacks for groups of integers modulo a prime. The special case of small subgroup attacks, that are based on submitting invalid elliptic curve points (more precisely, points that lie on a *different* curve) were, to our best knowledge, first described in [3]. The attack used in this paper is based on the attack sketched in [3], and explained in detail in [2]. It consists of two phases, an offline precomputation phase which must only be performed once for each elliptic curve parameters $\pi := (\mathbb{F}, a, b)$, and an online attack phase.

Offline precomputations. First, the attacker performs the following computations, which need to be performed only once for each particular choice of elliptic curve parameters $\pi := (\mathbb{F}, a, b)$ defining an elliptic curve group of order q.

1. Let $p_1, \ldots, p_n$ be the first n primes, such that $\prod_{i=1}^{n} p_i > q^2$. The attacker first computes integers $b_1, \ldots, b_n \in \mathbb{Z}_p$ such that $(\mathbb{F}, a, b_i)$ defines an elliptic curve of order q_i such that p_i divides q_i. To this end, the attacker sets $b_1 = \cdots = b_n = 0$, and repeats the following algorithm until $b_i \neq 0$ for all i.
   (a) Choose $b^* \xleftarrow{\$} \mathbb{Z}_p$ at random.
   (b) Count the number w of points on the curve $E_{(\mathbb{F},a,b^*)}$, by running the Schoof-Elkies-Atkin algorithm [7].
   (c) For each $i \in \{1, \ldots, n\}$, check if $p_i \mid w$. If this holds, set $b_i := b^*$.

   Note that the elliptic curve group defined by $(\mathbb{F}, a, b_i)$ has a small subgroup of order p_i, where all p_i are very small. Note also that it is sufficient to have $n \leq 2 \cdot \log_2 q$. By the prime number theorem, we may furthermore expect that the largest prime p_n has size about $p_n \approx n \cdot \ln n$. Thus, all primes $p_1, \ldots, p_n$ are very small, in the order of $O(\log q \cdot \log \log q)$. Assuming heuristically that the number of points w on the curve defined by $(\mathbb{F}, a, b^*)$ is distributed nearly uniformly over the interval $[q - 2\sqrt{q} + 1, q + 2\sqrt{q} + 1]$ (the interval given by the Hasse-Weil bounds [7]) for uniformly random $b^* \in \mathbb{Z}_p$, finding all b_i-values is expected to take about p_n iterations of the above algorithm.
   For example, if $q < 2^{193}$ is a 192-bit prime, then $n = 60$ and $p_n = 283$ is sufficient. For a 256-bit prime $q < 2^{257}$ we may have $n = 76$ and $p_n = 383$.
2. Next, the attacker determines points $G_1, \ldots, G_n$ such that G_i generates the subgroup of order p_i of the curve defined by $(\mathbb{F}, a, b_i)$.

For example, performing the above computations on a virtual machine running Ubuntu 12.04 LTS Server x64 with eight 2.3 GHz CPUs and 4 GB RAM takes about 90 min for the NIST P-192 curve, and about 5 h for the NIST P-256 curve, when both computations are started in parallel.

Online attack. In the online attack phase, the attacker interacts with a "target server". This server may, for example, be a TLS server implementing TLS-DH cipher suites. In order to describe the attack independently of a particular server (which would require to go into the details of the service provided by this server and its implementation), we describe the attack with "oracles" that capture the required behavior of a server in an abstract manner. We show later how to instantiate these oracles in practice.

In the sequel let $\mathcal{O}$ be an oracle that performs computations on a curve described by parameters $\pi := (\mathbb{F}, a, b)$. The oracle internally keeps a random secret $s \in \mathbb{Z}_q$. On input $G \in \mathbb{G}$, the oracle computes sG by applying the double-and-add algorithm, using the `DBL` and `ADD` procedures from Fig. 2. Finally, the oracle returns $[sG]_x$, the x-coordinate of point sG.[4]

Given the results $(b_i, G_i, p_i)_{1 \le i \le n}$ from the precomputation phase and oracle $\mathcal{O}$, the actual attack proceeds as follows.

1. First, $\mathcal{A}$ queries the oracle $\mathcal{O}$ n times, on inputs $G_1, \ldots, G_n$. Given G_i, the oracle computes and returns $[sG_i]_x$. Note that G_i does not lie on the curve defined by the "real" parameters $\pi := (\mathbb{F}, a, b)$, but on the curve defined by adversarially-chosen parameters $(\mathbb{F}, a, b_i)$. However, since the `DBL` and `ADD` procedures implemented by $\mathcal{O}$ are independent of b, the oracle will perform this computation correctly.
2. Next, $\mathcal{A}$ computes the points $t \cdot G_i$ for all $t \in \{0, 1, \ldots, (p_i + 1)/2\}$.[5] Then it defines s_i to the unique value t, such that $[sG_i]_x = [tG_i]_x$. Note that either $s_i \equiv s \mod p_i$, or $-s_i \equiv s \mod p_i$. Note also that $s_i^2 \equiv (-s_i)^2 \mod p_i$, thus, s_i^2 is a uniquely determined value.

   Since the group operations implemented by $\mathcal{O}$ are independent of elliptic curve parameter b, and we assume that the oracle does not check whether the given point G_i lies on the correct curve $E_{(\mathbb{F},a,b)}$, the oracle *implicitly* performs all computations on a different curve $E_{(\mathbb{F},a,b_i)}$ having a small subgroup of order p_i. This allows $\mathcal{A}$ to determine the unique value $s_i^2 \mod p_i$ for all $i \in \{1, \ldots, n\}$.
3. Finally, $\mathcal{A}$ computes the secret s by determining the unique integer $s \in \mathbb{Z}$ such that $s^2 < q^2$ and $s^2 \equiv s_i^2 \mod p_i$ for all $i \in \{1, \ldots, n\}$, by applying the Chinese Remainder Theorem (CRT) and the fact that the primes p_i have been chosen such that $\prod_{i=1}^{n} p_i > q^2$.

   The nice trick of computing with $s_i^2 \mod p_i$ is from [3]. It overcomes the issue that we learn either $s_i \mod p_i$ or $-s_i \mod p_i$, but without being able to test immediately which one is correct, by performing all CRT computations with the unique values $s_i^2 \mod p_i$ and finally computing the square root of the result s^2 over $\mathbb{Z}$.

[4] Note that keys in elliptic curve cryptography are often derived only from the x-coordinate of a point, which motivates this abstraction.

[5] In principle, this step can also be precomputed. However, we will later have to consider a slightly different setting (and thus a different oracle) where this precomputation is not possible, therefore we explain it here.

Remark 2. We will later describe an oracle $\mathcal{O}$ which takes as input G_i, and immediately returns $s_i^2 \mod p_i$ (instead of $[sG_i]_x$ as above). This essentially makes Step 2 of the online attack phase obsolete (in particular the computation of the values tG_i), and show how to realize this oracle in practice. Obviously, the above attack works identically with this oracle, by simply omitting Step 2. Describing the above attack with this particular oracle would, however, conceal the idea behind the invalid curve attack.

Remark 3. This attack can easily be prevented by replacing $\mathcal{O}$ with an oracle which checks whether a given point G lies on the "right" curve, that is, the defined by $\pi = (\mathbb{F}, a, b)$ before performing any computation. This is easy, by testing whether $y^2 \equiv x^3 + ax + b \mod p$. Note also that the test of group membership is relatively inexpensive, as it requires to compute only a small number of multiplications modulo p, which does not increase the complexity of computing sP significantly. Nevertheless, we will show the practical relevance of this attack.

3 Transport Layer Security

In the TCP/IP reference model, the TLS protocol is located between the transport layer and the application layer. Its main purpose is to protect insecure application protocols like HTTP or IMAP. It is also used as a building block in other protocols, like EAP-TLS authentication for WiFi networks.

The first (inofficial) version was developed in 1994 by Netscape, named *Secure Sockets Layer*. In 1999, SSL version 3.1 was officially standardized by the IETF Working Group and renamed to *Transport Layer Security* [8], version 1.0. Since then, two updates of the TLS specification were released, versions 1.1 [9] and 1.2 [10]. Version 1.3 is currently under development [11].

Cipher suites. TLS is rather a protocol framework than a fixed protocol that allows communicating parties to choose from a large number of different algorithms for the various cryptographic tasks performed in the protocol (key agreement, authentication, encryption, integrity protection). A *cipher suite* is a concrete selection of algorithms for all required cryptographic tasks. For example, a connection established with the cipher suite `TLS_RSA_WITH_AES_128_CBC_SHA` uses RSA-PKCS#1 v1.5 public-key encryption [12] to establish a key, and symmetric AES-CBC encryption with 128-bit key and SHA-1-based HMACs. Cipher suite `TLS_DHE_WITH_AES_128_CBC_SHA` uses the same symmetric algorithms, but establishes the key from a Diffie-Hellman key exchange with ephemeral exponents[6] and RSA-PKCS#1 v1.5 signatures [12] for authentication.

The TLS RFCs [8–10] and their extensions [4] specify a large number of different cipher suites. They can be divided into three large groups, depending on the key agreement algorithm used: In `TLS_RSA` cipher suites, the client

[6] That is, both communicating partners choose random exponents for each execution of the Diffie-Hellman protocol within TLS. Alternatively, there exist `TLS_DH` cipher suites, where the server uses a static exponent.

chooses a random PremasterSecret, encrypts it with the public RSA key of the server, and sends this cryptogram within the ClientKeyExchange message to the server. In TLS_DH and TLS_DHE, the Diffie-Hellman key exchange is used to establish the PremasterSecret. The difference between these two families is that in TLS_DH, the server DH share is static and contained in the server certificate, whereas in TLS_DHE, only a signature verification key is contained in the server certificate, and an ephemeral server DH share is contained in an additional ServerKeyExchange message. Both Diffie-Hellman variants can also be used with elliptic curves, in which case the substring "EC" is added to the cipher suite name. *In this paper, we only consider cipher suites from the* TLS_ECDH *family.*

3.1 The TLS-ECDH Handshake

At the beginning of each TLS session the *TLS Handshake* protocol is executed, to negotiate a cipher suite and cryptographic keys. In the following, we give a brief overview of the TLS_ECDH Handshake for all versions up to the latest version 1.2, in as much detail as required to explain our attack. Note that the sequence of messages exchanged in the handshake depends on the selected cipher suite.

Handshake overview. Let us first give an overview of the messages sent in the TLS Handshake. See also Fig. 1. A TLS handshake is initiated by a TLS client with a ClientHello message. This message contains information about the TLS version, a list of references to TLS cipher suites proposed by the client, and a client nonce r_C.

The server now responds with the messages ServerHello, Certificate, and ServerHelloDone. The ServerHello message contains a reference to a cipher suite, selected by the server from the list proposed by the client, the selected TLS version, and a server nonce r_S. The Certificate message contains an X.509 certificate with the server's public key; in case of TLS_ECDH the public key must be a point sP on the elliptic curve. The ServerHelloDone message indicates the end of this step.

The client responds with a ClientKeyExchange, which contains the ephemeral DH share of the client, i.e. a point qP on the curve, where q was chosen randomly, and P is the base point.

Finally, both parties send the ChangeCipherSpec and Finished messages. The former notifies the receiving peer that subsequent TLS messages will be protected (i.e. encrypted and MACed) using the newly negotiated cipher suite. The Finished message contains a MAC over all exchanged messages, and is necessary to protect against certain attacks (see [16]).

After the handshake has finished, the peers can start to exchange payload data, which are protected by the negotiated cryptographic algorithms and keys.

TLS_ECDH *cipher suites.* In TLS_ECDH, the ClientKeyExchange message contains the client's contribution qP to a EC-based Diffie-Hellman key exchange. Combined with the value sP from the server certificate, the PremasterSecret is

computed as $pms := [q(sP)]_x = [s(qP)]_x$. Note that only the x-coordinate of the resulting point is used as a `PremasterSecret`.

Using the TLS-PRF function, which is essentially a pseudorandom function based on an iterated HMAC, in a first step the `MasterSecret` ms is derived from pms:

$$ms := \mathsf{TLS\text{-}PRF}(pms; r_C, r_S, label_{ms}).$$

In a second step, the cryptographic keys and the `Finished` messages are derived using the `MasterSecret` as the key of the TLS-PRF:

$$keys := \mathsf{TLS\text{-}PRF}(ms; r_C, r_S, label_{keys}),$$
$$Fin := \mathsf{TLS\text{-}PRF}(ms; transcript)$$

Note that there is no explicit server authentication. The server authenticates implicitly, by being able to compute the `Finished` message correctly. This message depends on the `PremasterSecret`, thus the server must have been able to compute pms.

On client authentication via TLS. Note that we have described only server-authentication. It is in principle also possible to authenticate clients cryptographically in the TLS handshake, however, this would require *client certificates*. If an application requires client-authentication, then it is common to implement this by running an additional protocol over the established TLS channel, e.g. by transmitting a password. TLS is most commonly used with *server-only* authentication, therefore we focus on this setting.

4 Invalid Curve Attack on TLS-ECDH

In Sect. 2.2 we described an invalid curve attack on elliptic curve cryptosystem. In this section we will show how to obtain the required oracle responding with $s_i^2 \mod p_i$, given a point G_i on a curve $(\mathbb{F}, a, b_i)$ with a small subgroup of order p_i from a TLS server. We assume that this TLS server supports TLS-ECDH cipher suites. Moreover, the server does not validate whether a point sent by the client belongs to a specified curve or not, and implements the group law in a way which is "compatible" with both the real parameters $(\mathbb{F}, a, b)$ and the adversarially-chosen parameters $(\mathbb{F}, a, b_i)$. As explained above, the latter holds in particular if the server implements the standard double-and-add algorithm for multiplication of elliptic curve points with scalars.

The main difficulty in constructing such an oracle from a TLS server is that the server does not directly respond with a result of a multiplication sG. Instead, it uses this result internally to derive cryptographic keys, and expects a suitable TLS `ClientFinished` message. Thus, we will construct an oracle $\mathcal{O}$ which will establish several TLS connections to verify a guessed value sG_i, by sending `ClientFinished` messages. More precisely, given a point G_i and its order p_i prepared by the attacker $\mathcal{A}$, the oracle $\mathcal{O}$ proceeds as follows (see also Fig. 3):

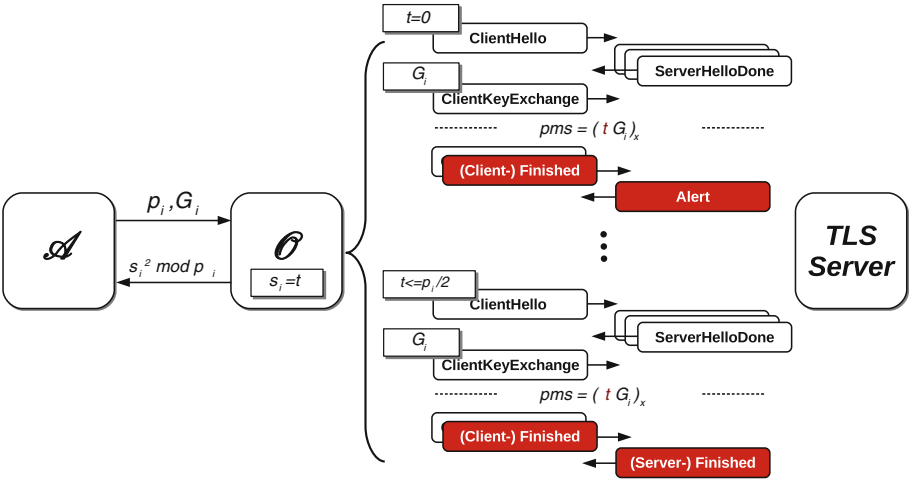

Fig. 3. Constructing an oracle $\mathcal{O}$ from a vulnerable TLS server supporting TLS-ECDH cipher suites.

1. $\mathcal{O}$ sets $t = 0$.
2. $\mathcal{O}$ starts a TLS handshake with a `ClientHello` message containing TLS-ECDH cipher suites (e.g., `TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA`). It receives TLS messages from the server and sends to the server a `ClientKeyExchange` message containing point G_i.
3. $\mathcal{O}$ guesses the $PremasterSecret$ and sets it to $pms = [tG_i]_x$. Based on the $PremasterSecret$, $\mathcal{O}$ computes the $MasterSecret$ and derives all keys needed for encryption and HMAC computations. $\mathcal{O}$ uses the derived keys to authenticate and encrypt the `ClientFinished` message.
4. If the TLS server accepts the `ClientFinished` message and responds with a `ServerFinished` message, the guessed $PremasterSecret$ was correct and it holds that $s \equiv \pm t \mod p_i$. $\mathcal{O}$ sets $s_i := t$ and responds with $s_i^2 \mod p_i$. Otherwise, if the server responds with a TLS alert message and terminates the connection, the guessed $PremasterSecret$ was incorrect. $\mathcal{O}$ increments t and proceeds with Step 2.

Note that $\mathcal{O}$ needs at most $p_i/2$ TLS handshake executions to get $s_i^2 \mod p_i$, and $p_i/4$ executions on average.

This oracle allows the attacker $\mathcal{A}$ to execute the full attack and recover server's private key. $\mathcal{A}$ first queries $\mathcal{O}$ n times, on inputs (G_i, p_i), where $i \in \{1, 2 \ldots, n\}$. It receives equations $s^2 \equiv s_i^2 \mod p_i$. Afterwards, $\mathcal{A}$ computes s^2 using the CRT and finally obtains the server's secret s.

5 Practical Evaluation

In this section we describe the invalid curve attacks on real implementations. To test the TLS implementations and libraries, we implemented a TLS client capable of sending invalid EC points in the `ClientKeyExchange` message, and complete a valid TLS handshake with a given `PremasterSecret`. In case an analyzed implementation was vulnerable to the attack, we used our TLS client to perform the complete attack. Otherwise, we analyzed why the attack was impossible. We conducted all the tests on a localhost, with a machine running on Xubuntu 14.10, with an Intel i7 processor (2.6 GHz).

5.1 Analyzed TLS Libraries

In order to use cryptographic libraries in Java dynamically, a system of cryptographic service providers was introduced. A cryptographic service provider "refers to a package or set of packages that supply a concrete implementation of a subset of cryptography features."[7] Java offers developers cryptographic providers, which are shipped directly with the Java installation (e.g., a SunEC provider for EC computation). The developers can however bind further providers like a Bouncy Castle provider to extend the default behavior of the installed providers.

For testing purposes, we set up a simple TLS server based on the *Java Secure Socket Extension (JSSE)*[8]. JSSE is used, for instance, in JBoss Application Server,[9] Apache Tomcat,[10] or Apache Camel framework.[11] We dynamically exchanged different cryptographic security providers to test their behavior while processing invalid EC points: Bouncy Castle, SunEC 1.8, and SunPKCS11-NSS 1.7. Further C/C++ libraries were tested with TLS test servers provided by these libraries.

- *Bouncy Castle Java 1.50.* Bouncy Castle[12] is a Java-based cryptography library, which can be bound to an implementation as a cryptographic provider. This library was heavily used for EC computations in Java 6, since Java 6 did not support EC by default. It can however also be used with further Java versions. In our work, we first tested Bouncy Castle 1.50 and then reevaluated our results with the 1.52 version.
- *MatrixSSL 3.7.1.* MatrixSSL is a C implementation designed specifically for small and embedded devices.[13]

[7] http://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html#Provider.
[8] http://docs.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html.
[9] http://jbossas.jboss.org/.
[10] https://tomcat.apache.org/.
[11] http://camel.apache.org/.
[12] https://www.bouncycastle.org/java.html.
[13] http://www.matrixssl.org/.

- *mbed TLS 1.3.10.* mbed TLS (formerly known as PolarSSL) is a lightweight C++ implementation also designed for small devices.[14]
- *OpenSSL 1.0.2a and LibreSSL 2.1.6.* OpenSSL is a cryptographic library with a TLS functionality.[15] LibreSSL is a fork of OpenSSL, created in 2014.[16] Our analysis revealed that the relevant EC implementation parts contain the same code, thus we treat them together as one library.
- *SunEC Security Provider 1.8.* SunEC is an Oracle Java security provider, which supports EC computations.[17] It is by default included in Oracle JDK 7 and 8, and in OpenJDK 8. In our tests, we used the SunEC provider distributed with Oracle JDK 1.8.0_40.
- *SunPKCS11-NSS Security Provider 1.7.* SunPKCS11-NSS is a Java security provider created as a wrapper over Mozilla's NSS library.[18] It is used as a default provider in OpenJDK 7 to support elliptic curves.
- *WolfSSL 3.4.6.* WolfSSL (formerly known as CyaSSL) is an embedded TLS library for small devices, written in C.

5.2 Attacks on Bouncy Castle

Analysis with our TLS client showed that a TLS server based on the Bouncy Castle library does not verify whether a given point lies on the right curve. For the point multiplication, the standard double-and-add algorithm is used. This allowed us to apply the attack described in Sect. 4 in a straightforward way.

Our evaluation with a secp256r1 elliptic curve revealed that the attacker needs about 3300 real server queries to get the private server key. In our localhost setup the online attack phase took about 155 s, see Table 2.

Table 2. Number of queries and time needed to execute the attack against a TLS server using the Bouncy Castle library in version 1.50. Note that a real attack over the Internet would last about ten to hundred times longer, depending on the server response times.

Elliptic curve	# of oracle queries	# of server queries	Duration [sec]
secp256r1	74	3300	155

We informed Bouncy Castle developers about this problem in their official developer mailing list.[19] It was patched one month after our disclosure, with the Bouncy Castle version 1.51. We are not sure whether our disclosure influenced the patch, since we got no official response.

[14] https://mbed.org/technology/mbed-tls/.
[15] https://www.openssl.org/.
[16] http://www.libressl.org/.
[17] http://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html#SunEC.
[18] https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS.
[19] http://bouncy-castle.1462172.n4.nabble.com/EC-Implementation-problems-td4657043.html.

5.3 Attacks on SunEC Security Provider

Our analysis of a TLS server using the SunEC security provider indicated that the SunEC provider is also vulnerable to the attacks described above. The server based on this provider processed invalid EC points and we were able to execute valid TLS handshakes. However, a full attack execution was not successful. Further analysis revealed that the SunEC provider introduced failures in the EC point multiplication, which resulted in wrong responses of the oracle constructed using the TLS server. Since the SunEC provider is implemented as a closed source, we needed to provide a black box analysis of the EC multiplication implementation.

Several tests with the EC computation showed that the probability of an invalid point multiplication depends on the order of the elliptic curve. More precisely, point multiplications on an elliptic curve group with order $p_i < 100$ returned a valid result with a probability of less than 60 %. Multiplications on elliptic curves with an order $p_i \approx 300$ returned a valid result with a probability of more than 90 %. Elliptic curves of an order $p_i \approx 1000$ computed correctly with a probability of about 98 %. See some exemplary results in Fig. 4, which depicts the SunEC computation correctness probability as a dependency of the elliptic curve order. The results were generated by applying 100 computations on 256 bit elliptic curves with random scalars.

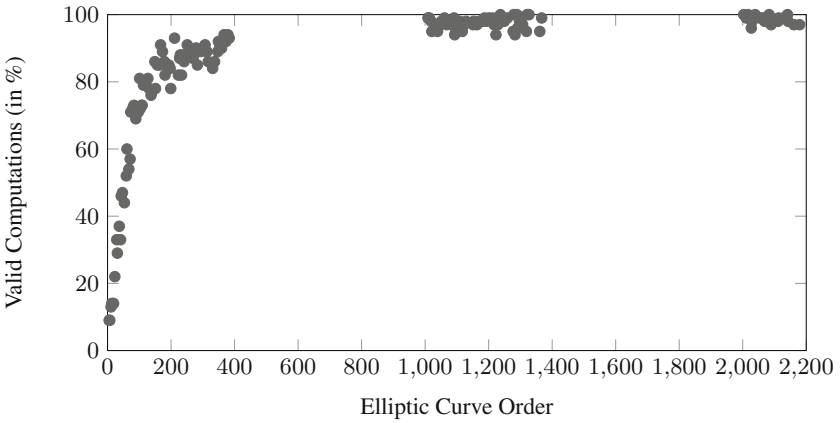

Fig. 4. Exemplary results showing dependency between the elliptic curve order and the percentage of valid EC computations executed by the SunEC provider: When working with custom elliptic curves with a small order ($p_i < 100$), only about one half of the computations were correct. This forced us to use elliptic curves with higher orders.

This is not a unique behavior of an EC implementation. A similar documented behavior of an invalid EC multiplication was observed in 2007 [18], when OpenSSL incorrectly multiplied specific points on a secp384r1 NIST curve. The reason was an incorrect handling of carry bits by the OpenSSL library. In our

tests, we were however not able to analyze the reason for the incorrect computation by the SunEC provider due to the fact that the source code is not publicly available.

The SunEC provider behavior forced us to use elliptic curves with an order $p_i > 1000$, where the probability of a valid point multiplication was about $\rho \approx 98\,\%$. This resulted in a success probability $\rho_s := \rho^n \approx 36\,\%$ for computing a valid server secret s, where $n = 50$ is the number of oracle queries to attack a server using the secp256r1 curve. In order to increase the chance of computing a valid secret, we adapted the algorithm as follows:

1. The attacker $\mathcal{A}$ sends to the oracle $(n + n')$ queries, where $\prod_{i=1}^{n} p_i > q^2$ and n' are additional attack queries.
2. $\mathcal{A}$ computes $\binom{n+n'}{n}$ possible values for the secret s.
3. $\mathcal{A}$ tests, which of the possible secrets is correct, such that the base point multiplied by the secret returns server public key sP.

Note that both the second and the third steps are offline steps, which can be executed after querying the server.

This adapted algorithm resulted in an overall success probability of

$$\rho_s := \sum_{i=0}^{n'} \binom{n+i}{n} \cdot \rho^{(n+n'-i)} \cdot (1-\rho)^i.$$

We could for example compute a valid server secret s with a probability $\rho_s \approx 75\,\%$ with $n' = 3$ additional attack queries.

Table 3. Number of queries and time needed to execute the attack against a TLS server using the SunEC security provider. Note that a real attack over the Internet would last about ten to hundred times longer, depending on the server response times.

Elliptic curve	# of oracle queries	# of server queries	Duration [sec]
secp192r1	40	14732	346
secp256r1	52	16897	412

In Table 3 we summarize our attack results. As can be seen, using elliptic curves of a higher order resulted in lesser oracle queries, but in higher number of total server queries (in comparison to the attacks on Bouncy Castle presented in Table 2). We could execute the attacks in less than 7 min, in our localhost setup.

We informed Oracle security team about this vulnerability. Oracle is going to provide a patch in the Oracle Critical Patch Update in July 2015.

6 Attack Impact and Countermeasures

Checking whether a given point lies on the correct curve is a simple and effective countermeasure against the attacks described in this paper, its computational

complexity is negligible in comparison to a full scalar multiplication of an elliptic curve point. The library providing elliptic curve point multiplication should therefore always validate whether the incoming point lies on the elliptic curve. Unfortunately, this seems not generally known to implementers of elliptic curve cryptography. Our attacks showed practical examples where this validation was omitted, and highlights that it is dangerous even in applications where the attack of [2] is not immediately applicable. The mentioned vulnerable implementations are already fixed or currently being fixed.

The described attack can be compared with the Heartbleed bug in the sense that the attack leaks the server's long-term private key to an attacker, and thus enables the attacker to impersonate the server in the future. However, we stress that TLS-ECDH cipher suite are less frequently used in practice than TLS-ECDHE or TLS-RSA ciphersuites, thus, the practical impact of these attacks is not as dramatic as the Heartbleed bug. Nevertheless, it is highly recommended to revoke and replace certificates used for static ECDH cipher suites in case the TLS server uses one of the vulnerable libraries or runs on a vulnerable Oracle JDK version, and supports TLS-ECDH cipher suites. This includes for example a JBoss Application Server, Apache Tomcat, or Apache Camel framework.

The attack on TLS is an important and particularly interesting special case. However, we stress that the omitted point validation in the considered libraries may also enable attacks on other protocols and applications beyond TLS. Thus, it is furthermore advisable to replace vulnerable elliptic curve libraries in any application using elliptic curve cryptography with secure ones, and to revoke and replace certificates for static ECDH cipher suites used in these applications.

A Further Analysis

In Table 4, we provide further analysis of secure TLS libraries and their EC computation processing. Our analysis furthermore includes the Bouncy Castle 1.52 library version, which contains a fix to the attack presented in this paper. We investigate whether the libraries use the standard double-and-add algorithm or a specific window method, where the point validation takes place (before or after point multiplication, or directly after point decoding), and what is the response of the TLS server.

As can be seen, most of the libraries use a window multiplication method and an explicit point validation function. An exception is the WolfSSL library, which does not verify whether the incoming EC point lies on the curve. We were able to send an arbitrary point in the `ClientKeyExchange` message and let the server compute a `PremasterSecret` using this point. However, the invalid curve attacks were not applicable, because the library uses a specific window multiplication method and this method depends on the curve parameter b of $\pi := (\mathbb{F}, a, b)$. We still recommend the developers to fix this issue and implement explicit point validation.

In case of the SunPKCS11-NSS security provider, we were not able to analyze the source code and find out which multiplication method was used or whether

Table 4. Analysis of secure TLS libraries and their processing of elliptic curve multiplication.

Bouncy Castle 1.52

Multiplication	Window method	Package `org.bouncycastle.math.ec` AbstractECMultiplier.multiply custom.sec.SecP256R1FieldElement.multiply
Point Validation	After multiplication	math.ec.ECAlgorithms.validatePoint
Handshake Termination	Fatal Alert, Internal Error	

MatrixSSL 3.7.1

Multiplication	Window method	crypto/pubkey/ecc.c: function eccMulmod
Point Validation	Point decoding	crypto/pubkey/ecc.c: function eccTestPoint
Handshake Termination	Fatal Alert, Decode Error	

mbed TLS 1.3.10

Multiplication	Window method	library/ecp.c: function ecp_mul_comb
Point Validation	Before multiplication	library/ecp.c: function ecp_check_pubkey_sw
Handshake Termination	Connection termination, no Alert message	

OpenSSL 1.0.2a (and LibreSSL 2.1.6)

Multiplication	Window method	crypto/ec/ec_mult.c: function ec_wNAF_mul
Point Validation	Point decoding	crypto/ec/ecp_oct.c: function EC_POINT_is_on_curve, invoked by ec_GFp_simple_oct2point
Handshake Termination	Connection termination, no Alert message	

SunPKCS11-NSS Security Provider 1.7

Multiplication	–	–
Point Validation	–	–
Handshake Termination	Fatal Alert, Internal Error Caused by: ....InvalidKeySpecException: Could not create EC public key at ....P11ECKeyFactory.engineGeneratePublic(P11ECKeyFactory.java:169)	

WolfSSL 3.4.6

Multiplication	Window method	wolfcrypt/src/ecc.c: function ecc_mulmod
Point Validation	No validation	–
Handshake Termination	Connection termination, no Alert message	

the point validation takes place. Our table thus just includes a visible stack trace provided by the tested TLS server.

References

1. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy, pp. 526–540, Berkeley, California, USA, 19–22 May 2013. IEEE Computer Society Press (2013)
2. Antipa, A., Brown, D., Menezes, A., Struik, R., Vanstone, S.: Validation of elliptic curve public keys. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 211–223. Springer, Heidelberg (2002)

3. Biehl, I., Meyer, B., Müller, V.: Differential fault attacks on elliptic curve cryptosystems. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 131–146. Springer, Heidelberg (2000)
4. Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., Moeller, B.: Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. Updated by RFCs 5246, 7027
5. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the rsa encryption standard PKCS #1. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 1–12. Springer, Heidelberg (1998)
6. Brumley, B.B., Barbosa, M., Page, D., Vercauteren, F.: Practical realisation and elimination of an ECC-related software bug attack. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 171–186. Springer, Heidelberg (2012)
7. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F. (eds.): Handbook of Elliptic and Hyperelliptic Curve Cryptography. Discrete Mathematics and its Applications (Boca Raton). Chapman and Hall/CRC Press, Boca Raton (2006)
8. Dierks, T., Allen, C.: The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176
9. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346 (Proposed Standard), April 2006. Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176
10. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176
11. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. draft-ietf-tls-tls13-04, January 2015
12. Kaliski, B.: PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational), March 1998. Obsoleted by RFC 2437
13. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press, Boca Raton (2007)
14. Lim, C.H., Lee, P.J.: A key recovery attack on discrete log-based schemes using a prime order subgroup. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 249–263. Springer, Heidelberg (1997)
15. McGrew, D., Igoe, K., Salter, M.: Fundamental Elliptic Curve Cryptography Algorithms. RFC 6090 (Informational), February 2011
16. Meyer, C., Schwenk, J.: SoK: lessons learned from SSL/TLS attacks. In: Kim, Y., Lee, H., Perrig, A. (eds.) WISA 2013. LNCS, vol. 8267, pp. 172–189. Springer, Heidelberg (2014)
17. Möller, B., Duong, T., Kotowicz, K.: This POODLE Bites: Exploiting the SSL 3.0 Fallback, September 2014. Technical report
18. Reimann, H.: Bn_nist_mod_384 gives wrong answers. openssl-dev mailing list #1593 (2007). http://marc.info/?t=119271238800004
19. Rizzo, J., Duong, T.: Here Come The ⊕ Ninjas, Ekoparty, May 2011