Remote Timing Attacks are Still Practical[*]

Billy Bob Brumley and Nicola Tuveri

Aalto University School of Science, Finland
{bbrumley,ntuveri}@tcs.hut.fi

Abstract. For over two decades, timing attacks have been an active area of research within applied cryptography. These attacks exploit cryptosystem or protocol implementations that do not run in constant time. When implementing an elliptic curve cryptosystem that provides side-channel resistance, the scalar multiplication routine is a critical component. In such instances, one attractive method often suggested in the literature is Montgomery's ladder that performs a fixed sequence of curve and field operations. This paper describes a timing attack vulnerability in OpenSSL's ladder implementation for curves over binary fields. We use this vulnerability to steal the private key of a TLS server where the server authenticates with ECDSA signatures. Using the timing of the exchanged messages, the messages themselves, and the signatures, we mount a lattice attack that recovers the private key. Finally, we describe and implement an effective countermeasure.

Keywords: Side-channel attacks, timing attacks, elliptic curve cryptography, lattice attacks.

1 Introduction

Side-channel attacks utilize information leaked during the execution of a protocol. These attacks differ from traditional cryptanalysis attacks since side-channels are not part of the rigorous mathematical description of a cryptosystem: they are introduced by implementation aspects and are not modeled as input and/or output of the cryptosystem. A timing attack is a side-channel attack that recovers key material by exploiting cryptosystem implementations that do not run in constant time: their execution time measured by the attacker is somehow state-dependent and hence key-dependent.

In light of these attacks, implementations of elliptic curve cryptosystems that execute in environments where side-channels are a threat seek to fix the execution time of various components in said implementation. Perhaps the most critical is that of scalar multiplication that computes the k-fold sum of a point with itself. Leaking any internal algorithm state during this computation can reveal information about some of the inputs, some of which should critically remain secret.

[*] Supported in part by the European Commission's Seventh Framework Programme (FP7) under contract number ICT-2007-216499 (CACE).

As a practical example of utilizing said key material, consider lattice attacks. Lattices are mathematical objects that have many uses in cryptography from cryptographic primitives to attacking schemes with partially known secret data. They are generally useful for finding small solutions to underdetermined systems of equations. Lattice methods are an effective endgame for many side-channel attacks: combining public information with (private) partial key material derived in the analysis phase, i.e., procured from the signal, to recover the complete private key. Repeatedly leaking even a small amount of ephemeral key material can allow these attacks to succeed at recovering long-term private keys.

Montgomery's ladder is a scalar multiplication algorithm that has great potential to resist side-channel analysis. The algorithm is very regular in the sense that it always executes the same sequence of curve and field operations, regardless of the value that a key bit takes. Contrast this with, for example, a basic right-to-left double-and-add scalar multiplication algorithm that only performs point additions on non-zero key bits.

This paper describes a timing attack vulnerability in OpenSSL's ladder implementation for elliptic curves over binary fields. The timings are procured by measuring the execution time of parts of the TLS handshake between OpenSSL's own TLS client and server where the server provides an ECDSA signature on a number of exchanged messages. We utilize this timing information to mount a lattice attack that exploits this vulnerability and recovers the ECDSA private key given a small number of signatures along with said timing data. We provide extensive experiment results that help characterize the vulnerability. Lastly, we propose, implement, and evaluate a simple and efficient countermeasure to the attack that proves effective.

The remainder of the paper is organized as follows. Section 2 reviews the concept of timing attacks and selective related literature. Section 3 contains background on elliptic curve cryptography and its implementation in OpenSSL. Section 4 identifies said vulnerability and describes all stages of the proposed attack. Section 5 contains the experiment and attack implementation results. We close in Section 6 with a discussion on countermeasures and draw conclusions.

2 Timing Attacks

P. Kocher gives a number of remarkably simple timing attacks in his seminal work [Koc96]. Consider a right-to-left square-and-multiply algorithm for exponentiation. If the exponent bit is a 1, the algorithm performs the assignments $B := B \cdot A$ then $A := A^2$. Otherwise, a 0-bit and the algorithm performs only the assignment $A := A^2$. The attacker chooses operand A hence its value in each iteration is known. To mount a timing attack, the attacker is tasked with finding input A that distinguishes former cases from the latter. This could be done by choosing A such that the former case incurs measurably increased execution time over the entire exponentiation yet the latter case does not. Varying the number of computer words in A could be one method to induce this behavior. Starting with the least significant bit, the attacker repeats this process to recover the key iteratively. In this manner, the attacker traces its way through the states of the exponentiation algorithm using the timings as evidence. The author gives further examples of software mechanisms that lead to timing vulnerabilities as well as attack experiment results. The work mostly concerns public key cryptosystems with a static key such as RSA and static Diffie-Hellman.

D. Brumley and D. Boneh [BB03,BB05] present ground breaking results, demonstrating that timing attacks apply to general software systems, defying contemporary common belief. They mount a timing attack against OpenSSL's implementation of RSA decryption based on (counteracting but exploitable) time dependencies introduced by the Montgomery reduction and the multiplication routines used by the OpenSSL implementation. The key relevant fact about the Montgomery reduction is that an extra reduction step may be required depending on the input, while for the multi-precision integer multiplication routines (heavily used in RSA computation) the relevant fact is that one of two algorithms with different performances (Karatsuba and schoolbook) is used depending on the number of words used to represent the two operands. Exploiting these two facts and adapting the attack to work even when using the sliding window modular exponentiation algorithm, the authors devise an attack that is capable of retrieve the complete factorization of the key pair modulus.

The authors mount a real-world attack through a client that measures the time an OpenSSL server takes to respond to RSA decryption queries during the SSL handshake. The attack is effective between two processes running on the same machine and two virtual machines on the same computer, in local network environments and in case of lightly loaded servers. The authors also analyze experiments over a WAN and a wireless link to evaluate the effects of noise on the attacks. Finally, they devise three possible defenses and as a consequence several cryptography libraries including OpenSSL feature RSA blinding by default as a countermeasure. As a tangible result of their work:

- OpenSSL issued[1] a security advisory;
- CVE assigned[2] the name CAN-2003-0147 to the issue;
- CERT issued[3] vulnerability note VU#997481.

3 Elliptic Curve Cryptography

In the mid 1980s, Miller [Mil85] and Koblitz [Kob87] independently proposed the use of elliptic curves in cryptography. Elliptic curves are a popular choice for public key cryptography because no sub-exponential time algorithm to solve discrete logarithms is known in this setting for well-chosen parameters. This affords Elliptic Curve Cryptography (ECC) comparatively smaller keys and signatures. For the purposes of this paper, it suffices to restrict to curves of the form

$$E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + a_2x^2 + a_6$$

[1] http://www.openssl.org/news/secadv_20030317.txt

[2] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0147

[3] http://www.kb.cert.org/vuls/id/997481

where $a_i \in \mathbb{F}_{2^m}$ and $a_2 = 1$ is common. NIST standardizes two types of curves for each $m \in \{163, 233, 283, 409, 571\}$:

1. a_6 is chosen pseudo-randomly: i.e., B-163.
2. $a_6 = 1$ and $a_2 \in \{0, 1\}$: Koblitz curves [Kob91], i.e., K-163.

With Intel's recent `pclmulqdq` carry-less multiplication instruction facilitating multiplication in $\mathbb{F}_2[x]$, curves over binary fields are likely to become the standard choice for high-speed ECC implementations in software.

3.1 Digital Signatures

We use the following notation for the ECDSA. The parameters include a hash function h and point $G \in E$ that generates a subgroup of prime order n. In fact $\#E = cn$ where $c \in \{2, 4\}$ for the standard curves considered in this paper. A common current choice for these would be roughly a 160-bit n, i.e., computations on B-163 or K-163. Parties select a private key d uniformly from $0 < d < n$ and publish the corresponding public key $D = [d]G$. To sign a message m, parties select nonce k uniformly from $0 < k < n$ then compute the signature (r, s) by

$$r = ([k]G)_x \bmod n \tag{1}$$

$$s = (h(m) + dr)k^{-1} \bmod n. \tag{2}$$

This work omits the details of signature verification as they are not particularly relevant here. The performance bottleneck for generating these signatures is the scalar multiplication in (1). Extensive literature exists on speeding up said operation: a description of one common method follows.

3.2 Scalar Multiplication

The speed of an ECC implementation is essentially governed by the scalar multiplication operation that, for an integer k and point $P \in E$, computes the point $[k]P$. There are many methods to carry out this computation, but we focus on the Montgomery power ladder, originally proposed for speeding up integer factorization using the elliptic curve method [Mon87, Sect. 10.3.1].

López and Dahab improve the algorithm efficiency for curves over binary fields [LD99]. Fig. 1 illustrates the main parts of the algorithm and is an excerpt from the implementation in OpenSSL 0.9.8o. The nested `for` loop is where the majority of the work takes place and performs one point doubling and one point addition to process one bit of k in each iteration; assume $k_i = 1$. The point addition formula, i.e. implemented in the `gf2m_Madd` function called in Fig. 1, is

$$(Z_0, X_0) = ((X_0 \cdot Z_1 + X_1 \cdot Z_0)^2, x \cdot Z_0 + (X_0 \cdot Z_1) \cdot (X_1 \cdot Z_0))$$

and point doubling, i.e. implemented in the `gf2m_Mdouble` function called in Fig. 1, is

$$(Z_1, X_1) = ((X_1 \cdot Z_1)^2, X_1^4 + a_6 \cdot Z_1^4).$$

An intriguing feature is that when $k_i = 0$, the same steps are performed: only the operands are transposed. That is, replacing Z_1 with Z_0 and X_1 with X_0 describes the above formulae for a zero bit. This means the cost per bit is fixed at an impressive six field multiplications, one involving a constant. For curves over binary fields, OpenSSL uses this algorithm as the default for any single scalar multiplication, e.g. in signature generation, and in fact iterates it twice for the sum of two scalar multiplications, e.g. in signature verification.

```
/* find top most bit and go one past it */
i = scalar->top - 1; j = BN_BITS2 - 1;
mask = BN_TBIT;
while (!(scalar->d[i] & mask)) { mask >>= 1; j--; }
mask >>= 1; j--;
/* if top most bit was at word break, go to next word */
if (!mask)
  {
  i--; j = BN_BITS2 - 1;
  mask = BN_TBIT;
  }

for (; i >= 0; i--)
  {
  for (; j >= 0; j--)
    {
    if (scalar->d[i] & mask)
      {
      if (!gf2m_Madd(group, &point->X, x1, z1, x2, z2, ctx)) goto err;
      if (!gf2m_Mdouble(group, x2, z2, ctx)) goto err;
      }
    else
      {
      if (!gf2m_Madd(group, &point->X, x2, z2, x1, z1, ctx)) goto err;
      if (!gf2m_Mdouble(group, x1, z1, ctx)) goto err;
      }
    mask >>= 1;
    }
  j = BN_BITS2 - 1;
  mask = BN_TBIT;
  }
```

Fig. 1. Montgomery's ladder scalar multiplication for curves over binary fields as implemented in OpenSSL 0.9.8o at `crypto/ec/ec2_mult.c`.

The ladder applied to ECC has numerous advantages: fast computation, no large memory overhead, and a fixed sequence of curve operations. This last feature is particularly attractive as a side-channel countermeasure: the following quote concisely captures this [HMV04, p. 103].

> Another advantage is that the same operations are performed in every iteration of the main loop, thereby potentially increasing resistance to timing attacks and power analysis attacks.

While this feature cannot be denied, the quoted authors duly qualify the statement with *potentially*: the side-channel properties are those of the algorithm implementation, not the algorithm itself. It should be noted that the ladder was originally proposed only for efficient computation. Its potential to resist side-channel analysis seems to be an unintentional consequence.

4 A Timing Attack

The ladder implementation in Fig. 1 introduces a timing attack vulnerability. Denote the time required to process one scalar bit and compute one ladder step as t: that is, one iteration of the nested `for` loop that performs the double and add steps. Said time is (reasonably) independent of, for example, any given bit k_i or the Hamming weight of k. On the other hand, consider the preceding `while` loop: its purpose is to find the index of the most significant set bit of k and optimize the number of iterations of the nested `for` loop. As a result, there are exactly $\lceil \lg(k) \rceil - 1$ ladder step executions and the time required for the algorithm to execute is precisely $t(\lceil \lg(k) \rceil - 1)$. This shows that there is a direct correlation between the time to compute a scalar multiplication and the logarithm of k.

This section describes an attack exploiting this vulnerability. The attack consists of two phases.

1. The attacker collects a certain amount of signatures and exploits the described time dependency to filter a smaller set of signatures. The signatures in the filtered set will have an high probability of being generated using secret nonces (k) having a leading zero bits sequence whose length is greater or equal to a fixed threshold.
2. The attacker mounts a lattice attack using the set of signatures filtered in the collection phase to recover the secret key used to generate the ECDSA signatures.

For this attack to succeed, we assume to be able to collect a sufficient amount of ECDSA signatures made under the same ECDSA key, and to be able to measure, with reasonably good accuracy, the wall clock execution time of each collected sign operation. For concreteness we focus on the NIST curve B-163, but the concepts can be more generally applied for any curve over a binary field.

4.1 Overview of the collection phase

To verify and evaluate the actual exploitability of the described time dependency for mounting a practical side-channel attack, we implemented two different versions of the collection phase that share the same basic structure and differ only for the sequence of operations used to perform a signature:

– a "local" attack, where the collecting process directly uses the OpenSSL ECDSA routines, accurately measuring the time required by each sign operation; this version models the collection phase in ideal conditions, where noise caused by external sources is reduced to the minimum;

- a “remote” attack, where the collecting process uses the OpenSSL library to perform TLS handshakes using the ECDHE_ECDSA suite; this version models a real-world use case for this vulnerability and allows to evaluate how practical the attack is over different network scenarios.

In general, regardless of the internal implementation, the sign routines of both versions simply return a signature, a measure of the time that was required to generate it, and the digest value fed to the sign algorithm. The collecting process repeatedly invokes the sign routine and stores the results in memory using a fixed-length binary tree heap data structure, where the weight of each element is represented by the measured time and the root element contains the maximum, using the following algorithm:

```
Heap h=Heap.new(s); //fixed size=s
from 1 to t:
  Result res=sign_rtn(dgst, privk);
  if ( !h.is_full() ):
    h.insert(res); //O(log n) time
  else if ( res.t < h.root().t ):
    h.root()<-res; //O(1) time
    h.percolate_down();//O(log n) time
  else:
    ; //discard res
```

With this algorithm we are able to store the smallest (in terms of time) s results using a fixed amount of memory and less than $O(t(1+\lg(s)))$ time in the worst case; the total number of signatures collected (t) and the size of the filtered set (s) are the two parameters that characterize the collection process. As we expect the fastest signatures to be related to nonces with an higher number of leading zeroes, we can use the ratio t/s to filter those signatures associated with leading zero bits sequences longer than a certain threshold.

Statistically for a random $1 \leq k < n$ with overwhelming probability the most significant bit will be set to zero since n for B-163 (and indeed many curves over binary fields) is only negligibly over a power of two. For the next leading bits the probability of having a sequence of zero bits of length j is equal to 2^{-j}. Hence if the total amount of collected signatures t is big enough, the set composed of the quickest s results should contain signatures related to nonces with leading zero bits sequences of length longer than $\lg(t/s)$, that are then fed to the lattice attack phase.

4.2 Collection phase in ideal conditions

This version of the collecting process was implemented to verify that the described time dependency is actually exploitable for mounting a side-channel attack. We directly invoked the OpenSSL ECDSA routines from within the collecting process to generate ECDSA signatures of random data, accurately measuring the time required by each sign operation.

The high resolution timings were taken using the `rdtsc` instruction provided in recent Pentium-compatible processors. As the host CPU used for testing was dual core and supported frequency scaling, to ensure accuracy of the measurements we disabled frequency scaling and forced the execution of the collecting process on just one core.

As the time needed to generate a signature does not depend on the value of the message digest, for simplicity and to speed up the experiments we chose to generate multiple signatures on the same message, precalculating the message digest just once to avoid generating a new random message for each signature.

The implemented sign routine takes as input the digest of the message to be signed and the private key, and returns the computed signature, the time required to compute the ECDSA signature and the number of leading zero bits in the nonce. The latter value is obviously not used to mount the actual attack: but is used only to verify the dependency between the execution time of the signature computation and the number of leading zero bits in the nonce.

4.3 Collection phase over TLS

This implementation of the collecting process was developed to show a relevant real-world use case for this vulnerability and to evaluate its practicality in different network scenarios.

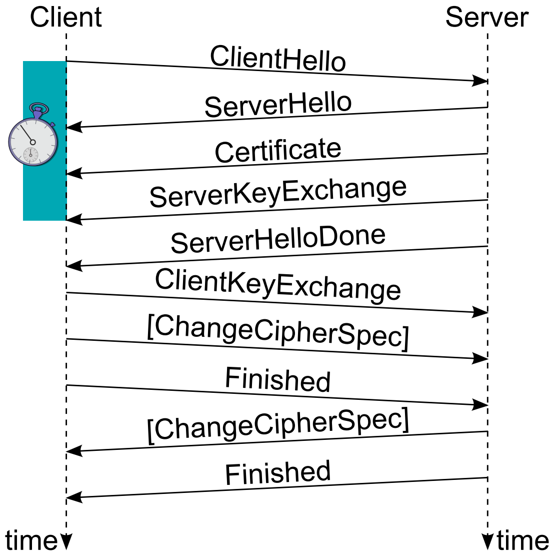

Fig. 2. TLS Handshake using the ECDHE_ECDSA suite described in RFC 4492.

Here the signatures collected are those generated during the TLS handshake using the ECDHE_ECDSA cipher suite illustrated by Fig. 2. We briefly high-light the relevant features of the messages exchanged during the portion of the handshake targeted by this attack, referring to RFC 4492 [BWBG$^+$06] for the normative and detailed technical description of the full protocol handshake:

- The Client initiates the handshake sending a ClientHello message to the Server; this is a regular TLS ClientHello message, proposing the ECDHE_ECDSA cipher suite and intended to inform the Server about the supported curves and point formats. This message contains a random nonce generated by the Client.
- The Server replies with a ServerHello message, selecting the proposed ECDHE_ECDSA cipher suite and using an extension to enumerate the point formats it is able to parse. This message contains a random nonce generated by the Server.
- The Server sends a Certificate message, conveying an ECDSA-signed certificate containing the ECDSA-capable public key of the Server, and possibly a certificate chain.
- The Server sends a ServerKeyExchange message, conveying the ephemeral ECDH public key of the Server (and the relative elliptic curve domain parameters) to the Client. This message is divided in two halves, the first one containing the Server ECDH parameters (namely the EC domain parameters and the ephemeral ECDH public key, consisting in an EC point) and the latter consisting of a digitally signed digest of the exchanged parameters. The digest is actually computed as SHA(ClientHello.random + ServerHello.random + ServerKeyExchange.params), and the signature is an ECDSA signature generated using the Server's private key associated with the certificate conveyed in the previous message.
- The handshake then continues, but other messages do not influence the implemented attack.

This version of the collecting process uses the OpenSSL library to perform an ECDHE_ECDSA TLS handshake every time a signature is requested. The sign routine creates a new TLS socket to the targeted IP address, configured to negotiate only connections using ECDHE_ECDSA and setting a message callback function that is used to observe each TLS protocol message. After creating the TLS socket the sign routine simply performs the TLS handshake and then closes the TLS connection. During the handshake the message callback inspect each TLS protocol message, starting an high resolution timer when the ClientHello message is sent and then stopping it upon receiving the ServerKeyExchange message, which is then parsed to compute the digest fed to the sign algorithm and to retrieve the generated signature.

In designing this attack we assumed to be unable to directly measure the actual execution time of the server-side signature generation, hence we are forced to use the time elapsed between the ClientHello message and the ServerKeyExchange message as an approximation. To assess the quality of this approximation the collecting process takes the private key as an optional argument. If provided, the message callback will also extrapolate the nonce used internally by the server to generate the signature and will report the number of leading zero bits in it.

Lastly, at first glance it seems possible that the Server's computation of its ECDHE key also influences the measured time. When creating an SSL context within an application, the default behavior of OpenSSL is to generate a key pair and buffer it for use before any handshake begins. This is done to improve efficiency. OpenSSL's internal `s_server` used in these experiments behaves accordingly, so in practice that step of the handshake does not affect the measured time since only one scalar multiplication takes place server-side during these handshake steps: that corresponding to the ECDSA signature. Applications can modify this behavior by passing options to the SSL context when creating it.

4.4 The Lattice Attack

Using lattice methods, Howgrave-Graham and Smart show how to recover a DSA key from a number of signatures under the same key where parts of the nonces are known [HGS01]. For completeness, a discussion on implementing the lattice attack follows. Observing j signatures, rearranging (2) yields j equations of the form

$$m_i - s_i k_i + d r_i \equiv 0 \pmod{n} \tag{3}$$

for $1 \leq i \leq j$ where here m_i are message digests to simplify notation. Using one such (3) to eliminating the private key yields $j-1$ equations of the form

$$k_i + A_i k_j + B_i \equiv 0 \pmod{n} \tag{4}$$

for $1 \leq i < j$ and some $0 \leq A_i, B_i < n$. From here, the equations in [HGS01] simplify greatly since all the known bits are in the most significant positions and are in fact all zeros: (4) should be considered the same as Equation 3 of [HGS01]. That is, express the nonces as $k_i = z'_i + 2^{\lambda_i} z_i + 2^{\mu_i} z''_i$ in their notation but all λ_i are zero setting all z'_i to zero and from the timings deducing all $\mu_i = 156$ (for example) setting all z''_i to zero, leaving z_i as the only unknown on the right where in fact $k_i = z_i$. This is nothing more than a rather laborious way of expressing the simple fact that we know $\lg(k_i)$ falls below a set threshold. Consider a j-dimensional lattice with basis consisting of rows of the following matrix.

$$\begin{pmatrix} -1 & A_1 & A_2 & \dots & A_{j-1} \\ 0 & n & 0 & \dots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & \dots & \dots & \dots & n \end{pmatrix}$$

From here, the implementation uses the Sage software system to produce a reduced basis for this lattice using the LLL algorithm [LLL82], orthogonalize this basis by the Gram-Schmidt process, and approximate the closest vector problem given input vector $(0, B_1, B_2, \dots, B_{j-1})$ using Babai rounding [Bab86]. This hopefully finds the desired solutions to the unknown portions of the k_i.

Figure 3 contains experiment results of running the lattice attack with different parameters based on B-163, assuming upper bounds on $\lceil \lg(k_i) \rceil$ of $\mu_i \in \{156, 157\}$. The x-axis is the signature count (j) and the y-axis the observed lattice attack success probability. It shows that as the amount of known key material decreases (μ_i increases), this mandates an increase in the lattice dimension j (the number of such required signatures j increases), and the approximations are less likely to hold. To effectively apply this as part of the timing attack, on one hand the lower we set μ_i the less likely it is that a k_i will satisfy the bound and more signatures must be collected. On the other hand, collecting more signatures increases the probability of error in the measurements, i.e., incorrectly inferring a given signature with a low timing has a k_i that satisfies the bound and the lattice attack is more likely to fail. An interesting property of this particular lattice attack is that in fact μ_i does not feature in the equations used to populate the basis matrix. In practice, this means that even if some k_i does not satisfy the bound there is still a chance the attack will succeed.

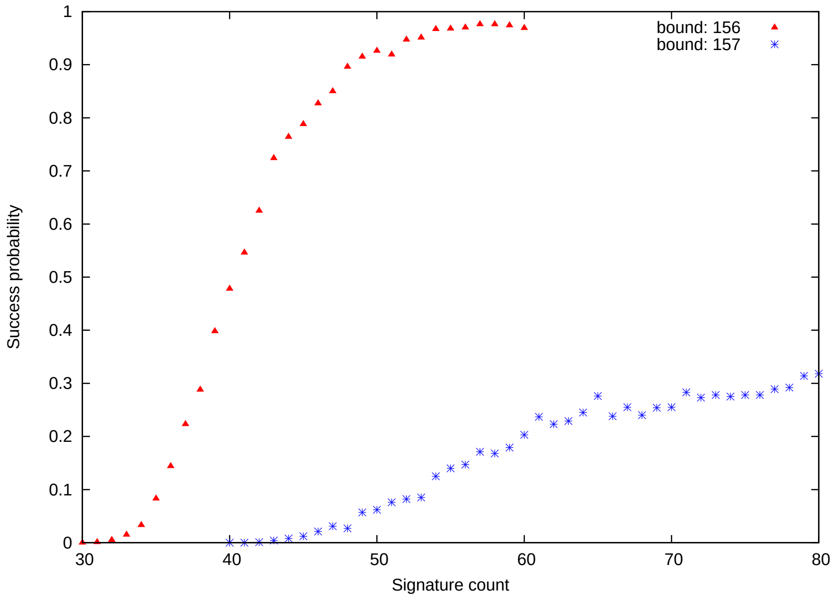

Fig. 3. Selective lattice attack parameters and observed success probabilities.

5 Results

5.1 Collection phase parameters

Using the first implementation of the collecting process we were able to empirically verify the dependency between the length of the leading zero bits sequence in the nonce and the execution time of the signature operation. Figure 4 compares the distributions of the execution time required by signatures generated using nonces with different leading zero bit sequence lengths. In the lattice attack notation, seven leading zero bits corresponds to $\mu_i = 156$ and six leading zero bits $\mu_i = 157$.

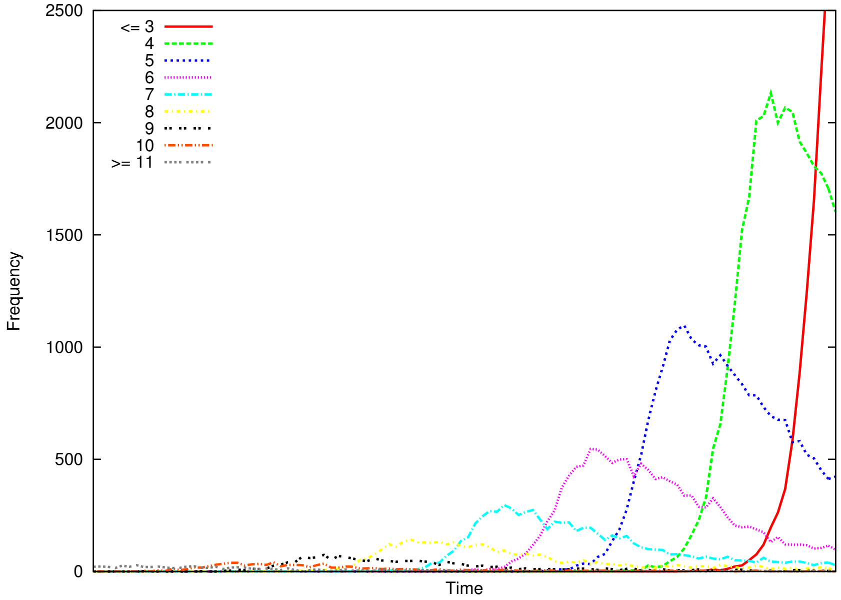

Fig. 4. Dependency between number of leading zero bits and wall clock execution time of the signature operation.

We then evaluated the effectiveness of the method used for filtering the signatures related to nonces with longer leading zero bit sequences by using different values for the collection phase parameters. The lattice attack phase, which takes in input the output of the collecting process, determines the size of the filtered set and the minimum length of the leading zero bit sequence of the nonce associated with the signature. Fixing the filtered set size to 64 signatures and varying the total number of signatures collected by the collecting process, we evaluated the number of “false positives” over multiple iterations, i.e., those signatures in the filtered set generated using nonces whose leading zero bit sequence length is below the threshold determined by the tuning of the lattice attack phase. Table 1 summarizes the obtained results and shows that the effectiveness of the filtering method may be adjusted varying the t/s ratio.

The number of “false positives” in the filtered set is an important parameter of the attack. The lattice attack phase has a higher success probability if all the signatures used to populate the matrix satisfy the constraint on the number of leading zero bits. But as mentioned in Sec. 4, even in the presence of limited “false positives” the lattice attack still succeeds with a small probability.

Table 1. Observed results of the local attack.

Collected signatures count (t)	4096	8192	16384
Filtered set size (s)	64	64	64
Average "false positives" count	17.92	1.48	0.05

Consulting Fig. 3, setting the threshold on the minimum number of leading zero bits to 7 we only needed 43 valid signatures to successfully perform the lattice attack with high probability. Naïvely, this allows up to 21 "false positives" in the filtered set obtained from the collecting process using the lattice attack in a more fault-tolerant way:

```
Signatures[] filtered_set; // <-- collection_phase()
EC_point known_pubkey; // <-- server certificate

while(True)
{
  tentative_privkey=lattice_attack(filtered_set[0:43]);
  tentative_pubkey=generate_pub_key(tentative_priv_key);
  if ( tentative_pub_key == known_pubkey )
    break;  // successfully retrieved the priv key
  randomly_shuffle(filtered_set);
}
```

What follows is a rough estimate for the number of required lattice attack iterations in the presence of "false positives". The number of iterations, and thus the computation time, needed to correctly retrieve the private key is inversely proportional to the probability of selecting a subset of the filtered set without "false positives":

$$\Pr[\text{subset without ``false positives''}] = \frac{\binom{64-e}{43}}{\binom{64}{43}}$$

where e is the number of "false positives" in the filtered set, 43 is the size of the subsets, 64 is the size of the filtered set, the numerator is the number of possible subsets without "false positives" in the filtered set, and the denominator is the number of possible subsets in the filtered set. Figure 5 shows how this probability varies with e.

5.2 Remote attack

We used the described "remote" implementation of the collecting process to attack TLS servers over two different network scenarios. As a reference server we used the OpenSSL `s_server` configured to emulate a TLS-aware web server using an ECDSA-capable private key. In theory every server using the OpenSSL ECDSA implementation to support ECDHE_ECDSA TLS should be vulnerable.

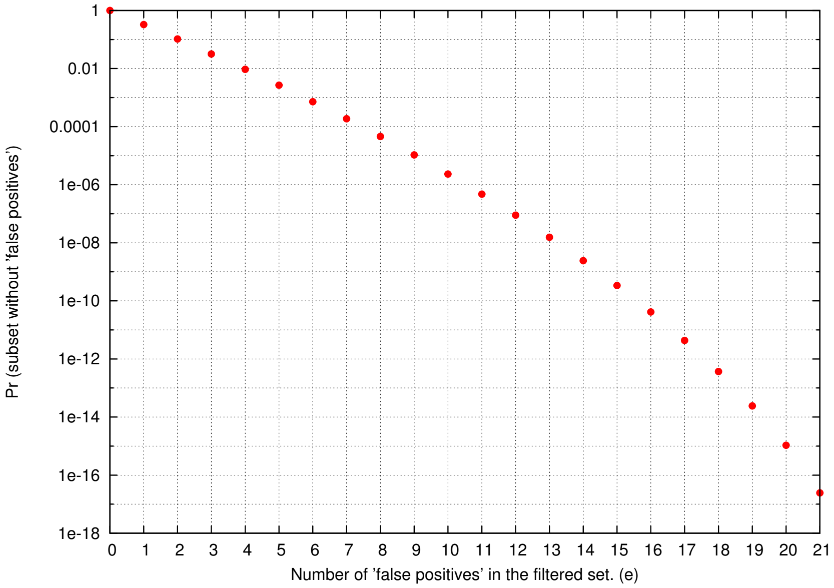

Fig. 5. Probability of selecting a subset without "false positives" in a filtered set with e "false positives".

The first scenario we considered consists of a collecting process running on the same host as the server process: the messages are exchanged over the OS TCP/IP stack using the localhost address on the loopback interface. In this scenario we successfully retrieved the server private key, even repeating the tests using different private keys, randomly generated using OpenSSL itself, and targeting both the OpenSSL 0.9.8o and 1.0.0a versions of the reference server.

Table 2 shows that, even if unable to directly measure the execution time of the signature computation, using the measure of the time elapsed between the ClientHello and the ServerKeyExchange messages as an approximation and tuning the total number of collected signatures, the attacker is able to filter a set of signatures with a low average of "false positives".

We also note that in the "remote" attack, only the collection phase is performed online, as the lattice attack phase does not require the attacker to exchange messages with the attacked server, and that even collecting a total of 16384 signatures is not particularly time consuming, requiring just few minutes.

Table 2. Observed results of the remote attack over the loopback interface.

Collected signatures count (t)	4096	8192	16384
Filtered set size (s)	64	64	64
Average "false positives" count	17.06	4.01	0.90

Once verified that the attack was practical when run over the loopback interface on the same host of the attacked server, we performed the same attack in a slightly more complex network scenario: the attacker collects the signatures from a host connected to the same network switch of the server. The tests were run between two hosts residing in the same room in time frames with reasonably low network loads, trying to minimize the noise introduced by external causes in the time measures.

From Table 3 we see that the time dependency is still observable: the average rates of "false positives" in the filtered sets increases, but from Fig. 5 this is still easily within reach. The lattice attack can take hours to run in this case, but the work is offline and can be distributed. In some cases we achieved success in only a few minutes. We also note that in this particular network environment we cannot arbitrarily decrease the "false positives" rate by increasing the parameter t, as already with $t = 16384$ the average number of "false positives" starts to increase.

Table 3. Observed results of the remote attack over a switched network segment.

Collected signatures count (t)	4096	8192	16384
Filtered set size (s)	64	64	64
Average "false positives" count	19.40	8.96	11.81

This demonstrates the feasibility of the attack in a remote scenario. Naturally individual results will vary due to different network characteristics. The attack success rate decreases dramatically with the increase of "false positives". Regardless, a vulnerability exploitable to perform a successful attack from the same host where the targeted server is run poses a threat even for remote attacks. For example, in virtual hosting or cloud computing scenarios an attacker may be able to obtain access to run code on the same physical machine hosting the target server, as suggested by [RTSS09].

6 Conclusion

This paper identifies a timing attack vulnerability in OpenSSL's implementation of Montgomery's ladder for scalar multiplication of points on elliptic curves over binary fields. This is used to mount a full key recovery attack against a TLS server authenticating with ECDSA signatures. Ironically, in the end it is the regular execution of the ladder that causes this side-channel vulnerability: for example, a dependency on the weight of k (that might leak from, say, a simple binary scalar multiplication method) seems much more difficult to exploit than that of the length of k that led to full key recovery here.

The work of D. Brumley and D. Boneh [BB03,BB05] and this work are related in that both exploit implementation features that cause variable time execution, and that both demonstrate full key recovery in both local and remote scenarios. However, the fundamental difference with the former is that the attacker can leverage well-established statistical techniques and repeat measurements to compensate for noise because the secret inputs are not changing, i.e., the RSA exponent. Contrasting with the latter, the secret inputs are always distinct, i.e., the nonces in ECDSA. The former is a stronger attack than the latter in this respect.

Lastly, a brief discussion on countermeasures follows. One way to prevent this attack is by computing $[k]G$ using the equivalent value $[\hat{k}]G$ where

$$\hat{k} = \begin{cases} k + 2n & \text{if } \lceil \lg(k+n) \rceil = \lceil \lg n \rceil, \\ k + n & \text{otherwise.} \end{cases}$$

With $\#\langle G \rangle = n$ then $[k]G = [\hat{k}]G$ holds and the signature remains valid. This essentially changes $\lceil \lg(\hat{k}) \rceil$ to a fixed value. We implemented this approach as a patch to OpenSSL and experiment results show that applying said padding thwarts this particular attack and does not entail any performance overhead to speak of. Note that the scope of this paper does not include microarchitecture attacks for which additional specific countermeasures are needed.

This work further stresses the importance of constant time implementations and rigorous code auditing, adding yet another entry to an already long list of cautionary tales surrounding cryptography engineering.

References

[Bab86] L. Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[BB03] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[BB05] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[BWBG$^+$06] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS). RFC 4492 (Informational), May 2006. Updated by RFC 5246.

[HGS01] Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Des. Codes Cryptography*, 23(3):283–290, 2001.

[HMV04] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.

[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *Math. Comp.*, 48(177):203–209, 1987.

[Kob91] Neal Koblitz. CM-curves with good cryptographic properties. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer, 1991.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[LD99] Julio López and Ricardo Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In Çetin Kaya Koç and Christof Paar, editors, *CHES*, volume 1717 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 1999.

[LLL82] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261(4):515–534, 1982.

[Mil85] Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. Comp.*, 48(177):243–264, 1987.

[RTSS09] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

A Countermeasure as OpenSSL Source Code Patch

```
--- openssl-0.9.8o-orig/crypto/ecdsa/ecs_ossl.c 2009-12-01 19:32:16.000000000 +0200
+++ openssl-0.9.8o-patch/crypto/ecdsa/ecs_ossl.c 2010-12-10 15:10:56.000000000 +0200
@@ -144,6 +144,16 @@
  }
  while (BN_is_zero(k));

+#ifdef ECDSA_POINT_MUL_NO_CONSTTIME
+ /* We do not want timing information to leak the length of k,
+  * so we compute G*k using an equivalent scalar of fixed
+  * bit-length. */
+
+ if (!BN_add(k, k, order)) goto err;
+ if (BN_num_bits(k) <= BN_num_bits(order))
+ if (!BN_add(k, k, order)) goto err;
+#endif /* def(ECDSA_POINT_MUL_NO_CONSTTIME) */
+
  /* compute r the x-coordinate of generator * k */
  if (!EC_POINT_mul(group, tmp_point, k, NULL, NULL, ctx))
  {
```