Recovering OpenSSL ECDSA Nonces Using the Flush+Reload Cache Side-channel Attack

No Author Given

No Institute Given

Abstract.

1 Introduction

Elliptic curve cryptography [6,7] is a collection of public key cryptographic methods that rely on the computational intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP). In a nutshell, given an elliptic curve over a finite field and two points on the curve G and H, the ECDLP is to find the scalar k such that $H = kG$.

Elliptic curve cryptography offers a higher encryption strength per key-bit than similar methods based on discreet logarithms over finite field or on number factoring. Consequently, elliptic curves cryptography uses significantly shorter keys and offers faster operations than other methods, contributing to the rising popularity of elliptic curves cryptography.

The Elliptic Curve Digital Signature Algorithm (ECDSA) [1, 4, 9] is a standard digital signature algorithm based on elliptic curves. The core operation of the ECDSA algorithm is multiplying a point on the elliptic curve by a randomly or pseudo-randomly chosen secret nonce. The confidentiality of the nonce is paramount for the security of the algorithm. Past research indicates that partial exposure of nonce bits can be exploited for efficient attacks on the secret key [2, 11].

OpenSSL [13] is a cryptographic software package that implements ECDSA. When using elliptic curves over a binary field $\mathbb{F}_{2^m}$, OpenSSL uses the Montgomery ladder [5, 8] algorithm for multiplying the point by the nonce. One of the advantages of the Montgomery ladder is that it has regular behaviour, performing the same sequence of operations for each nonce bit, irrespective of the value of the bit. This regular behaviour makes it more resilient to side-channel attacks [5, 12].

While the operations performed by the algorithm are regular, their targets depend on the value of the bits of the nonce. To apply the operations to the respective targets, the OpenSSL implementation uses a conditional branch based on the value of the bit. By tracing this branch an attacker can recover the values of the nonce bits and, consequently, break the cryptosystem.

In this paper we present our use the Flush+Reload cache side-channel attack [16] to trace the branch in the OpenSSL implementation. Flush+Reload relies on a security weakness in the IA-32 and X86-64 architectures that allows processes to monitor other processes read and execute access to shared memory pages. Our attack program monitors access to both arms of the conditional branch and uses the information collected from these probes to reconstruct the nonce.

The paper also presents new information on the limitation of the Flush+Reload attack. We discuss spatial limitations, affecting the distance between multiple probes, and temporal limitations, affecting the probe resolution.

⋆ Analysis of partial nonce exposure?

The rest of this paper is organised as follows. The next section presents background information on elliptic curves, ECDSA, the Montgomery ladder and the Flush+Reload attack. Section 3 describes our attack on the OpenSSL implementation of ECDSA. The results of the attack are analysed in Section 4. We discuss the implications of the attack and suggest techniques for mitigation in Section 5. Section 6 presents related research.

2 Preliminaries

2.1 ECDSA

The ElGamal Signature Scheme is the basis of the US 1994 NIST standard, Digital Signature Algorithm (DSA). The ECDSA is the adaptation of one step of the algorithm from the multiplicative group of a finite field to the group of points on an elliptic cure. The main benefit of using this group over field elements is smaller parameters for the same security level as mentioned above. ***

Parameters: An elliptic curve E defined over a finite field $\mathbb{F}_q$; a point $G \in E$ of large prime order n (generator of the group of points of order n). Paramaters chosen as such are generally believed to offer a security level of $\frac{n}{2}$ given current knowledge and technologies. Parameters are recommended to be generated following [10]. The field size q is usually taken to be a large, odd prime or a power of 2. The implementation of OpenSSL uses both prime fields and $q = 2^m$ though the results in this paper relate to the binary field case.

Public-Private Key pairs: the private key is an integer d, $1 < d < n-1$ and the public key is the point $Q = dG$. Calculating the private key from the public key requires solving the ECDLP, which is known to be hard in practice for the correctly chosen parameters. The most efficient algorithms currently know which solve the ECDLP have a square root run time in the size of the group, hence the aforementioned security level.

Suppose Bob, with public-private Key pair $\{d_B, Q_B\}$, wishes to send a signed message m to Alice, he follows the following steps:

1. Using an approved hash algorithm, compute $e = Hash(m)$, take $\bar{e}$ to be the leftmost n bits of e. ***check fips is leftmost or mod***
2. Randomly select $k \leftarrow_R \mathbb{Z}_n$ with $1 < k < p-1$ and $(k, p-1) = 1$.
3. Compute the point $(x, y) = kG \in E$. ***using montgomery ladder or otherwise***

4. Take $r = x \mod n$; if $r = 0$ then return to step 2.
5. Compute $s = k^{-1}(z + rd_B) \mod n$; if $s = 0$ then return to step 2.
6. Bob sends (m, r, s) to Alice.

The message m is not necessarily encrypted, the contents may not be secret, but a valid signature gives Alice strong evidence that the message was indeed sent by Bob. She verifies that the message came from Bob by

1. checking that all received parameters are correct, that $r, s \in \mathbb{Z}_n$ and that Bob's public key is valid, that is $Q_b \neq \mathcal{O}$ and $Q_B \in E$ is of order n.
2. Using the same hash function and method as above, compute $\bar{e}$.
3. Compute $\bar{s} = s^{-1} \mod n$.
4. Find the point $(x, y) = \bar{e}\bar{s}G + r\bar{s}Q_B$.
5. Verify that $r = x \mod n$ otherwise reject the signature.

Step 2 of the signing algorithm is of vital importance, inapproproate reuse of the random integer is what lead to the highly publicised breaking of PS3 signature scheme implementation. Knowledge of the random value k leads to knowledge of the secret key as all values (m, r, s) can be observed by an eavesdropper, $\bar{e}$ can be found from m, $r^{-1} \mod n$ can be easily found from n, and if k is discovered then an adversary can find Bob's secret key through the simple calculation

$$d_B = (sk - \bar{e})r^{-1}.$$

2.2 The Montgomery Ladder

Scalar multiplication is a common operation in cryptography and in a number of incidences (such as the multiplication by the secret, randomly generated element required in ECDSA), the scalar is intended to remain secret. This scalar multiplication is most efficiently performed using a square-and-multiply method (or the related Right-to-left method) as outlined in Algorithm 1

Double-and-add methods, though efficient, are vulnerable to simple power analysis. The addition law for points on Weirstrass curves is not complete, that is, the computation of $P + Q$ differs between the cases $P = Q$ and $P \neq Q$. Consequently, by examining the power consumption of the computation it is possible to distingush when the if loop is executed and hence when a bit of n is 0.

As described by Montgomery in [8]

2.3 The Flush+Reload attack

3 Attacking OpenSSL ECDSA

OpenSSL is one of the most common open-source cryptographic libraries. It provides a set of cryptographic services, including public and shared key encryption algorithms and public key signature algorithms.

OpenSSL's implementation of ECDSA uses the Montgomery ladder algorithm for scalar multiplication on the Elliptic Curve. We use this implementation to demonstrate that naïve implementations of the Montgomery ladder are susceptible to the Flush+Reload attack.

Listing 1.1 shows the relevant section of the implementation of the Montgomery ladder in OpenSSL version 1.0.1e. The bits of the multiplication scalar are stored in the word array `scalar->d`. The outer loop, at lines 268 to 286 traverses over the words representing the scalar. The inner loop, at lines 271 to 284 traverses the bits in each word. Line 273 tests the bit. For each bit the implementation executes a group add followed by a group double. If the bit is set, the implementation uses lines 275 and 276. For clear bits it uses lines 280 and 281.

Listing 1.1. OpenSSL Implementation of the Montgomery Ladder

```
268 for (; i >= 0; i--)
269 {
270     word = scalar->d[i];
271     while (mask)
272     {
273         if (word & mask)
274         {
275             if (!gf2m_Madd(group, &point->X, x1, z1, x2, z2, ctx))
                        goto err;
276             if (!gf2m_Mdouble(group, x2, z2, ctx)) goto err;
277         }
278         else
279         {
280             if (!gf2m_Madd(group, &point->X, x2, z2, x1, z1, ctx))
                        goto err;
281             if (!gf2m_Mdouble(group, x1, z1, ctx)) goto err;
282         }
283         mask >>= 1;
284     }
285     mask = BN_TBIT;
286 }
```

As the listing demonstrates, the Implementation is very regular. For each bit, the implementation executes exactly the same sequence of operations. The only differences between set and clear bit are the lines that invoke these operations. Our attack exploits this difference. The attack probes the cache lines used by OpenSSL to identify the branch taken by the `if` statement in line 273. From this information we can deduce the value of the bit and by repeating the attack throughout the execution of the scalar multiplication we can recover most of the bits of the nonce.

The mapping of source lines to cache lines in our build of OpenSSL is depicted in Diagram 1. The machine code created from source lines 273 to 282 covers the virtual memory address range 0x0812130C to 0x081213e8. This range spans four cache lines, marked A, B, C and D. The attacks observes the sequence of cache access made by the processor when executing the code.

$\star$ required, actual and observed cache access sequences

Fig. 1. Mapping from Source Code to Memory

The `if` statement at line 273 is executed for each bit. The code of this statement is in cache line A, hence this line is accessed when processing of a bit starts. For set bit, the processing continues with source line 275, which maps to cache lines A and B. The actual call to the group add function occurs at address 0x08121347. (See mark in Diagram 1.) After a delay for computing the group add, execution continues in cache line B to process the return value and to invoke the group doubling function. The group doubling function returns to cache line B and execution leaves the `if` body at cache line D.

Hence, the sequence of cache line accesses required for a set bit is: A, B, *add*, B, *double*, B, D. Similarly, for a clear bit, the sequence is: A, C, *add*, C, D, *double*, D.

Processor optimisations may add otherwise unrequired cache line accesses. The spatial prefetching optimisation [3] pairs adjacent cache lines and tries to bring both cache lines when there is a miss on one of the pair's line. For example, when there is a cache miss on cache line A, the spatial prefetcher may attempt to prefetch cache line B and vice versa.

Another optimisation that can cause additional cache line access is speculative execution [14]. With speculative execution, the processor follows both arms of a conditional branch before evaluating the condition. When the condition is evaluated, the processor commits to the pre-processed computation of the correct arm, disposing of the computation done for the other arm. In the case of OpenSSL this means that even before evaluating the bit, the processor may start processing both line 275 and line 280, triggering memory loads from cache lines A, B and C.

4 Results

- ⋆ Test architecture
  - ⋆ Curve - sect571k1
  - ⋆ Operation times
  - ⋆ Choice of slot size
  - ⋆ Example of results
  - ⋆ Number of lost bits

5 Discussion

- ⋆ LLL attack
  - ⋆ Expected number of observed signatures to break key
  - ⋆ Smaller keys are more resilient
    [15]
  - ⋆ Mitigation, Dan Bernstein NaCL

6 Related Work

7 Conclusions

References

1. American National Standards Institute: ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm. (1999)
2. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In Atluri, V., Diaz, C., eds.: Computer Security - ESORICS 2011. Volume 6879 of Lecture Notes in Computer Science., Springer-Verlag (2011) 355–371
3. Intel Corporation: Intel 64 and IA-32 Architecture Optimization Reference Manual. (April 2012)
4. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security **1**(1) (August 2001) 36–63
5. Joye, M., Yen, S.M.: The Montgomery powering ladder. In Kaliski, Jr., B.S., Koç, Ç.K., Paar, C., eds.: Cryptographic Hardware and Embedded Systems—CHES 2002. Volume 2523 of Lecture Notes in Computer Science., Springer-Verlag (2003) 291–302
6. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**(177) (January 1987) 203–209
7. Miller, V.S.: Use of elliptic curves in cryptography. In Williams, H.C., ed.: Advances in Cryptology - CRYPTO'85. Volume 218 of Lecture Notes in Computer Science., Springer-Verlag (1986) 417–426
8. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177) (January 1987) 243–264
9. National Institute of Standards and Technology: Digital Signature Standard, FIPS Publication 186-2. (2000)
10. National Institute of Standards and Technology: FIPS PUB 186-4 Digital Signature Standard (DSS). (2013)
11. Nguyen, P.Q., Shparlinski, I.E.: The insecurity of the elliptic curve digital signature algorithm with partially known nonces. Designs, Codes and Cryptography **30**(2) (September 2003) 201–217
12. Okeya, K., Kurumatani, H., Sakurai, K.: Elliptic curves with the Montgomery-form and their cryptographic applications. In: Public Key Cryptography. Volume 1751 of Lecture Notes in Computer Science., Springer-Verlag (2000) 238–257
13. OpenSSL. `http://www.openssl.org`
14. Uht, A.K., Sindagi, V.: Disjoint eager execution: An optimal form of speculative execution. In: Proceedings of the 28th International Symposium on Microarchitecture, Ann Arbor, Michigan, United States (November 1995) 313–325
15. Walter, C.D.: Longer keys may facilitate side channel attacks. In Matsui, M., Zuccherato, R.J., eds.: Selected Areas in Cryptography. Volume 3006 of Lecture Notes in Computer Science., Springer-Verlag (2004) 42–57
16. Yarom, Y., Falkner, K.: Flush+reload: a high resolution, low noise, l3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448 (2013) `http://eprint.iacr.org/`.

Input: Point P, scalar n, k bits
Output: Point nP
$Q \leftarrow \mathcal{O}$
for i *from* k *to* 0 **do**
    $Q \leftarrow 2Q$ **if** n_i *= 0* **then**
        $Q \leftarrow Q + P$
    end
end

Algorithm 1: Double-and-Add Point Multiplication

Input: Point P, scalar n, k bits
Output: Point nP
$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
for i *from* k *to* 0 **do**
    if n_i *= 0* **then**
        $R_1 \leftarrow R_0 + R_1$
        $R_0 \leftarrow 2R_0$
    else
        $R_0 \leftarrow R_0 + R_1$
        $R_1 \leftarrow 2R_1$
    end
end

Algorithm 2: Montgomery Ladder Point Multiplication