Working title: Extending the capabilities of the FLUSH+RELOAD Cache Side-channel Attack on OpenSSL using LLL

November 4, 2013

Abstract

1 Introduction

In the work of Yarom and Benger [16] some of the issues related to using the attack when implementations used the sliding window for scalar multiplications instead of the Montgomery ladder... in this work we bypass these issues by using the LLL based techniques [12] to recover the nonce.

1.1 Related Work

This paper is not only a fantastic read but the authors are just tops too! [16]

2 Background

In this section we present the relevant general background information about the attack and also the specific information required to understand the context of the example attack.

2.1 ECDSA

The ElGamal Signature Scheme [5] 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 curve. The main benefit of using this group as opposed to the multiplicative group of a finite field is that smaller parameters can be used to achieve the same security level [8, 9] due to the fact that the current best known algorithms to solve the discrete logarithm problem in the finite field are sub-exponential and those used to solve the ECDLP are exponential. See Balasubramanian and Koblitz [2], Adleman and Demarrais [1] and developments thereof for more details.

Parameters: An elliptic curve E defined over a finite field $\mathbb{F}_q$; a point $G \in E$ of a large prime order n (generator of the group of points of order n). Parameters chosen as such are generally believed to offer a security level of $\sqrt{n}$ given current knowledge and technologies. Parameters are recommended to be generated following the Digital Signature Standard [11]. 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 $q = 2^m$; 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 currently known algorithms for solving the ECDLP have a square root run time in the size of the group [4, 15], hence the aforementioned security level.

Suppose Bob, with private-public 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 ℓ bits of e (where $\ell = \min(\log_2(q)$, bitlength of the hash)).

2. Randomly select $k \leftarrow_R \mathbb{Z}_n$.

3. Compute the point $(x,y) = kG \in E$.

4. Take $r = x \bmod n$; if $r = 0$ then return to step 2.

5. Compute $s = k^{-1}(\bar{e} + rd_B) \bmod n$; 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 $\bar{s} = s^{-1} \bmod n$.

4. Find the point $(x,y) = \bar{e}\bar{s}G + r\bar{s}Q_B$.

5. Verify that $r = x \bmod n$ otherwise reject the signature.

Step 2 of the signing algorithm is of vital importance, inappropriate reuse of the random integer led to the highly publicised breaking of Sony PS3 implementation of ECDSA. Knowledge of the random value k, a.k.a. the *ephemeral key* or the *nonce*, leads to knowledge of the secret key. All values (m,r,s) can be observed by an eavesdropper, $\bar{e}$ can be found from m, $r^{-1} \bmod n$ can be easily computed from n and r, and if k is discovered then an adversary can find Bob's secret key through the simple calculation

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

Our attack targets Step 3 of the OpenSSL implementation of ECDSA.

2.2 Scalar multiplication using wNAF

Scalar multiplication is a common operation in cryptography and in a number of incidences (such as step 3 of ECDSA) the scalar is intended to remain secret. This scalar multiplication is most efficiently performed using a double-and-add method (or the related right-to-left method) as outlined in Algorithm 1.

```
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 scalar multiplication

Exponentiation (point scaler multiplication for ECC) is a common operation in cryptographic protocols but also one of the most expensive, much effort has been spent improving the speed of exponentiation algorithm; see [6] for a survey of exponentiation methods.

For the ECDSA as implemented using OpenSSL there are two aspects which can be exploited for: first, the scalar multiplication is performed on the same point every time; second, addition of ellitic cure points in the commonly used Weierstraß form costs the same as subtraction. Both properties are exploited by wNAF.

2.2.1 wNAF

As the scalar multiplication of step 3 of ECDSA is performed on the same point G at every execution (the random nonce changes), by calculating and storing some small multiples of G the scalar multiplication can be computed at a lower cost.

When using a window of size w, $2^w - 1$ extra points are stored and the precomutation cost is $2^{w-2} - 1$ point additions and one point doubling. For example, the window size used by OpenSSL is $w = 3$ so the points $\{3G, 5G, 7G\}$ are computed and the set $\{-7G, -5G, -3G, -G, G, 3G, 5G, 7G\}$ are used in the computation of kG. In algorithm 2 the conversion of the scalar k to the Non-Adjacent From (NAF) is given. The NAF is so named as there are no two non-zero k_i which are adjacent. The first line in the `if` statement will set k_i to be one of $\{\pm 1, \pm 3, \pm 5, \pm 7\}$ (for the OpenSSL implementation), this determines which of the precomputed points are used in the addition operation of the `if` statement in algorithm 3, which computed kG.

```
**Input:** scalar $k$
**Output:** $k$ in wNAF: $k_0, \ldots, k_{i-1}$
$i \leftarrow 0$
while $k > 0$ do
    if $k \mod 2 = 1$ then
        $k_i \leftarrow \min(|[k \mod 2^w] - 2^w|, |k \mod 2^w|)$
        $k = k - k_i$
    else
        $k_i = 0$
    end
    $k = k/2$
    $i+ = 1$
end
```

Algorithm 2: Conversion to Non-Adjacent Form

```
**Input:** scalar $k$ in wNAF $k_0, \ldots, k_{i-1}$, Points
$\{-7G, -5G, -3G, -G, G, 3G, 5G, 7G\}$
**Output:** $kG$
$Q \leftarrow \prime$
for $j$ from $i-1$ downto $0$ do
    $Q = 2Q$
    if $k_j ! = 0$ then
        $Q+ = k_j G$
    end
end
```

Algorithm 3: Computation of kG using wNAF

In the application of FLUSH+RELOAD by Yarom and benger [16], exploited the spy's ability to identify which arm of the `if` statement is exploited when scalar multiplication is computed using Montgomery multiplication, as explained in the following section. The work presented in this paper extends the capability of the attack in [16].

2.3 The FLUSH+RELOAD attack

The FLUSH+RELOAD attack, described by Yarom and Falkner [17], exploits a weakness in the IA-32 and X86-64 processor architectures, which allows processes to manipulate the cache of other processes.

Using the attack, a spy program can trace or monitor memory read and execute access of a victim program to shared memory pages. The spy program only requires read access to the shared memory pages, hence pages containing binary code in executable files and in shared libraries are susceptible to the attack. By monitoring the victim access to specific locations in these pages, the spy program learns when the victim executes the code in the monitored memory locations. From this information the spy program can infer information on the data processed by the victim.

Yarom and Falkner, in [17], and Yarom and Benger, in [16], use the FLUSH+RELOAD attack to retrieve the secret key from the GnuPG RSA decryption, and retrieve the nonce used in the ECDSA and hence the signer's private key, respectively.

The spy program monitors the phases of the square-and-multiply exponentiation [6] used by GnuPG and OpenSSL. As these phases depend on the values of the bits of the exponent, monitoring them allows the spy program to recover the secret exponent.

The attack operates by dividing time into slots. At the beginning of a time slot, the spy program flushes the monitored memory line from the cache of the processor. At the end of the slot, the spy program loads data from the memory line. Loading data from cached memory lines is significantly faster than loading them from memory. Hence, by measuring the time it takes to load the data, the spy program can know whether the line is cached or not. As the line is flushed at the beginning of the slot, having it cached at the end indicates that the processor accessed the line during the time slot.

In both of these works, the monitored branching of code by conditional statements statements would completely reveal the secret information as the condition tested for a set bit, the only alternative being a clear bits. When the implementation uses wNAF the condition is on the scalar k in non-adjacent form, if the element k_i is non-zero then although it is possible to detect that the `if` statement is executed, we do not immediately have the value of the non-zero k_i (for OpenSSL could be any of $\{\pm 1, \pm 3, \pm 5, \pm 7\}$).

As mentioned above, one characteristic of a number in non-adjacent form is that no two non-zero k_i which are adjacent, there are (at least) $w-1$ ($= 2$ for OpenSSL) zeros between the non-zero values. This is due to the fact that the subtraction of k_i from k in the `if` statement in algorithm 2 "wipes out" the lower bits of k.

3 Attacking OpenSSL

OpenSSL is one of the most commonly used open-source cryptographic libraries. It provides a set of cryptographic services, including both public key and symmetric encryption algorithms, and public key signature algorithms. OpenSSL's implementation of ECDSA uses one of two methods for performing the scalar multiplication; either the Montgomery ladder, fully analysed in [16], or sliding window. We demonstrate below that implementations using the sliding window method are equally vulnerable to the FLUSH+RELOAD attack. In

3.1 Extracting information

Listing 1 shows the relevant section of the implementation of the wNAF in OpenSSL version 1.0.1e. ***The bits of the multiplication scalar are stored in the word array `scalar->d`***, where the word size is defined by the architecture, e.g. 32 bits for the IA-32 architecture and 64 bits for the X86-64 architecture.

Listing 1: OpenSSL implementation of the wNAF sliding window

```
for (; i >= 0; i--)
{
    *** insert code
}
```

As the listing demonstrates.... ***non-zero bits cause the code in lines X to execute***, our spy program uses the FLUSH+RELOAD technique to monitor the execution of the `if` statement in line X. We know that one of the eight stored points was added, but not which one.

FLUSH+RELOAD monitors execution by placing probes on shared memory lines. For the attack to recover the bit values, it must distinguish between memory lines access sequences resulting from set bits and those resulting from clear bits. Achieving this depends on several factors: the mapping of source code to memory lines, the sequence of accesses to these memory lines when executing the code and FLUSH+RELOAD's ability to accurately capture the sequences.

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

Hence, the sequence of memory 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*.

Due to the limited temporal resolution of FLUSH+RELOAD, the attack can observe the order of memory accesses only if they are sufficiently separated in time. Hence, in the case of OpenSSL, the attack can only observe the order of memory accesses if they are separated by a call to a group operation. For example, when the bit is set, the attack cannot decide whether the access to memory line *A* precedes or follows the access to memory line *B*. Similarly, when observed by FLUSH+RELOAD, memory accesses issued after the group double are merged with those issued at the start of processing the following bit. Figure **??** shows the observable memory accesses when processing a set bit followed by a clear bit.

The diagram also shows memory accesses issued by processor optimisations. These optimisations pre-load memory lines into the cache to reduce the time the program waits for these lines. For example, when the processor uses speculative execution [13], it 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 memory lines *A*, *B* and *C*.

Another optimisation that can cause additional memory line access is spatial prefetching [7]. The processor pairs adjacent memory lines and tries to bring both memory lines into the cache when there is a miss on one of the pair's lines. For example, when there is a cache miss on memory line *A*, the spatial prefetcher may attempt to prefetch memory line B and vice versa.

Consequently, as demonstrated in Fig. **??**, the memory lines accessed between computing the group add and the group double can be used for recovering the value of the bit. Probing any of lines A and B gives a positive indication of set bits. Probing either line C or D gives a positive indication of clear bits. For our attack we probe memory lines B and D.

Three limitations of the FLUSH+RELOAD attack affect its ability to capture the sequence of memory accesses. The first is the attack temporal resolution which affects its ability to determine the order of accesses performed within a short time from each other. The second limitation is the possibility of an overlap between the memory access and the probe which may result in the attack missing the access. The third limitation is the result of the interaction between the FLUSH+RELOAD attack and the processor optimisation of cache use. In particular, the spatial prefetching optimisation implies that the attack cannot be used to probe two cache lines that form a pair, because probing one of the lines in a pair triggers a prefetch of the other.

For OpenSSL, the attack resolution should be sufficiently high for the attack to be able to distinguish between memory accesses done before and after each bit and those done between the group add and group double operations of each bit. This can be achieved by setting the time slot size to be less than the time it takes the victim to calculate the group double. As group double calculations are faster than group add calculations, this ensures that the probed memory lines are flushed when the victim computes the group add to be probed when the victim computes the group double.

The probability of an overlap, like the attack resolution, depends on the length of the time slot. Longer time slots mean that the portion of time during which the spy probes is smaller and, therefore, the probability of an overlap is lower.

As predicted by Walter [14], smaller keys are more resilient to the attack. With smaller keys, group operations are shorter, forcing shorter time slots. The shorter time slots lead to an increased probability of an overlap and with it of missing bits.

Missing memory accesses not only prevents the spy program from recovering the value of bits. It may also result in the spy program losing the bit position in the scalar multiplication process. To protect against this possibility, our attack also probes the first and last memory lines of the `gf2m_Mdouble` function. Probing these lines provides the spy program with additional information on the operation of the victim and facilitates recovering the position of captured scalar bits.

3.2 Full Nonce Recovery

Given the higher proportion of missing bits, BSGS method as used in [**?**] is not viable.

In [**?**] the relationship with the HNP was noticed and the idea developed in [12].

With prob $p = 2^{-n+1}$ we obtain the n least significant bits. Then require $\frac{256}{n}$ signatures, thus by observing $\frac{256}{n2^{n-1}}$ signatures, we have enough to use the LLL based techniques [12], ***implementation details, timings.***

The next section describes the details of our experimentation with the attack and its results.

4 Experimental Setup and Results

To test the FLUSH+RELOAD attack on OpenSSL we used an HP Elite 8300 running Fedora 18. As the OpenSSL package shipped with Fedora does not support ECC, we used our own build of OpenSSL 1.0.1e. To facilitate the mapping from source lines to memory addresses we built OpenSSL with debugging symbols. In real attack settings, the attacker will need to reverse engineer [3] the OpenSSL library. For the experiment we used the OpenSSL `sect1571r1` curve. (NIST Binary-Curve B-571 [11].)

With the selected curve, group add operations take 23,612 cycles on average. The first group double operation takes 6,552 cycles on average, whereas further group double operations take 11,962 cycles. Based on the discussion in Section 3, we picked a slot length of 10,240 cycles.

5 Discussion

Implications

Even if EC optimisations are present, the nature of the atttack would be able to distinguish bla bla...

Mitigation

6 Conclusions and future work

This is the end of the world as we know it!

7 acknowledgements

The authors wish to thank Dr Katrina Falkner for the advice and support.

This research was performed under contract to the Defence Science and Technology Organisation (DSTO) Maritime Division, Australia.

[10]

References

[1] L. M. Adleman and J. Demarrais. A Subexponential Algorithm for Discrete Logarithms over all Finite Fields. *Mathematics of Computation*, 61(203):1–15, 1993.

[2] R. Balasubramanian and N. Koblitz. The Improbability that an Elliptic Curve has Subexponential Discrete Log Problem under the Menezes - Okamoto - Vanstone Algorithm. *Journal of Cryptology*, 11(2):141–145, 1998.

[3] Teodoro Cipresso and Mark Stamp. Software reverse engineering. In Peter Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, chapter 31, pages 659–696. Springer, 2010.

[4] Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Improving the parallelized pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.

[5] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[6] Daniel M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, April 1998.

[7] Intel Corporation. *Intel 64 and IA-32 Architecture Optimization Reference Manual*, April 2012.

[8] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[9] Victor S. Miller. Use of Elliptic Curves in Cryptography. In H. C. Williams, editor, *Advances in Cryptology - Crypto '85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer, 1985.

[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`.

[11] National Institute of Standards and Technology. *FIPS PUB 186-4 Digital Signature Standard (DSS)*, 2013.

[12] Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2):201–217, September 2003.

[13] Augustus K. Uht and Vijay Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 313–325, Ann Arbor, Michigan, United States, November 1995.

[14] Colin D. Walter. Longer keys may facilitate side channel attacks. In Mitsuru Matsui and Robert J. Zuccherato, editors, *Selected Areas in Cryptography*, volume 3006 of *Lecture Notes in Computer Science*, pages 42–57. Springer-Verlag, 2004.

[15] Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In Stafford E. Tavares and Henk Meijer, editors, *Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 1998.

[16] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack, 2013.

[17] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448, 2013. `http://eprint.iacr.org/`.