Noname manuscript No.
(will be inserted by the editor)

Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack

Yuval Yarom · **Naomi Benger**

the date of receipt and acceptance should be inserted later

Abstract We illustrate a vulnerability introduced to elliptic curve cryptographic protocols when implemented using a function of the OpenSSL cryptographic library. For a given implementaion using an elliptic curve E over a binary field with a point $G \in E$, our attack can recover the majority of the bits of a scaler k when kG is computed using the OpenSSL implementation of the Montgomery ladder. For the Elliptic Curve Digital Signature Algorithm (ECDSA) the scalar k is intended to remain secret. Our attack recovers the scalar k and thus the secret key of the signer and would therefore allow unlimited forgeries. This is possible from snooping on only one signing process and requires computation of less than one second on a quad core desktop when the scalar k (and secret key) is around 571 bits.

1 Introduction

Elliptic curve cryptography (ECC) [16, 14] includes a number of public-key cryptographic protocols whose security relies on the computational intractability of the Elliptic Curve Discrete Logarithm Problem (ECDLP): Given an elliptic curve over a finite field and two points on the curve G and H, find the scalar k such that $H = kG$.

ECC offers a higher encryption strength per key-bit than related methods whose security is reliant on the hardness of computing discrete logarithms in a finite field or factoring the product of large primes. Consequently, ECC uses significantly shorter keys and offers faster operations than other methods, contributing to its rising popularity.

The Elliptic Curve Digital Signature Algorithm (ECDSA) [12, 18, 2] is a standard digital signature algorithm implemented using elliptic curves. One core operation of the ECDSA algorithm, as in many ECC protocols, is the scalar multiplication of a point on the elliptic curve by a pseudo-randomly generated 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 [19, 5].

OpenSSL [21] 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 [17, 13] algorithm to compute kG, the scalar multiplication of a publically known point G by the secret nonce k. 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 [13, 20].

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 of the FLUSH+RELOAD cache side-channel attack [27] to trace the branch in the OpenSSL implementation.

The FLUSH+RELOAD attack exploits 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 spy program monitors access to both arms of the conditional branch and uses the information collected from these probes to reconstruct the nonce. This attack is a threat to the security of any cryptographic protocol implemented using the OpenSSL scalar multiplication method when the scalar is intended to remain secret.

Yuval Yarom · Naomi Benger
The University of Adelaide, E-mail: yval@cs.adelaide.edu.au, naomi.benger@adelaide.edu.au

In this paper we illustrate the efficiency of the atttack by analysing ECDSA and recovering the secret key using only one signature at very little computational cost (in both time and memory). This attack is applicable when the mallicious party has access to the memory of the targeted device. This is a reasonable assumption as could be the case when using, for example, a multi-user operating system, co-hosted virtual machines in a cloud computing environment or a computer victim to malware.

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. The results of this paper support the findings of [25] that longer keys render a cryptographic algorithm more vulnerable to side-channel analysis.

The rest of this paper is organised as follows: The next section presents background information on ECDSA, the Montgomery ladder and the FLUSH+RELOAD attack followed by a short discussion of related research. 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.

2 Preliminaries

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. We also discuss the results of previous, related work.

2.1 ECDSA

The ElGamal Signature Scheme [8] 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 as opposed to the multiplicative group of a finite field is that smaller parameters for the same security level [14,16] due to the fact that the current best known algorithms to solve the discrete logarithm problem in the finite field are subexponential and theose used to solve the ECDLP are exponential, see [3,1], 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 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 $\sqrt{n}$ given current knowledge and technologies. Parameters are recommended to be generated following [18]. 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 algorithms currently know which solve the ECDLP have a square root run time in the size of the group [26,7], 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 ℓ bits of e (where $\ell = \min(\log_2(q)$,bitlength of the hash)).
2. Randomly select $k \leftarrow_R \mathbb{Z}_n$ with $1 < k < p-1$ and $(k, p-1) = 1$.
3. Compute the point $(x,y) = kG \in E$.
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, inappropriate reuse of the random integer is what lead to the highly publicised breaking of Sony PS3 implementation of ECDSA. Knowledge of the random value k 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, $r^{-1} \mod n$ can be easily found from n, and if k is discovered then an adversary can find Bob's secret key through the simple calculation

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

Step 3 of the signing algorithm is the stage targeted by this attack, when implemented using OpenSSL's montgomery ladder.

2.2 The Montgomery Ladder

Scalar multiplication is a common operation in cryptography and in a number of incidences (such as step 2 of ECDSA as above), 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.

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

Double-and-add methods, though efficient, are vulnerable to side-channel attacks. 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, 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 [17], the Montgomery ladder is presented in Algorithm 2. It differs from Algorithm 1 in that both a doubling and addition of points occurs at each step, regardless of the bit value of k. Thus, the Montgomery ladder thwarts side channel attacks which measure the computation at each bit and thus determine if an addition operation was executed. The branching in Algorithm 2 controls which point is doubled and where the addition of points is stored. The attack presented in this paper uses a different method to determine which branch of the algorithm was executed. This method is described in the following section.

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 Scalar Multiplication

2.3 The FLUSH+RELOAD attack

FLUSH+RELOAD is a recently developed cache side-channel attack [27]. The attack 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 by 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.

The spy program described in [27] uses the FLUSH+RELOAD attack to retrieve the secret key from the GnuPG RSA decryption. The spy program monitors the phases of the square-and-multiply exponentiation used by GnuPG. 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.

When the victim memory access overlaps the spy measurement, the spy will miss the access. Consequently, increasing the time slot length reduces the portion of time the spy spends in measurement and with it the probability of missing access. On the other hand, the spy is unable to distinguish between multiple accesses to the same memory line in a single time slot. Consequently, increasing the time slot reduces the attack's resolution. Hence choosing the length of the time slot presents a tradeoff between the attack resolution and the probability of missing a memory access.

2.4 Related Work

There have been a number of publications addressing the security issues of digital signatures when partial information is leaked, including [10, 9, 19].

In [9] algorithms for solving the ECDLP using the additional information of some consecutive bits of the private key are presented. These algorithms outperform the currently best known methods of solving the ECDLP without the extra information or using exhaustive search on the remaining

key space. In this work we do not focus on the ECDLP, we instead used leaked information about the ephemeral keys.

The attacks [10, 19] rely on having obtained a relatively small number of bit of the ephemeral keys used for many signatures and then use the LLL method [15] to solve the related hidden number problem to find the secret value. The attack of [19], for example, given a group of order around 160 bits the probabilistic algorithm in would obtain the secret key using 23 signatures (assuming independent and uniformly at random selected messages) in polynomial time, using only 7 consecutive, least significant leaked bits of the nonce (relying on some reasonable assumptions). Each of these assumes only a small fraction of k is recovered. The main contribution of this work is to illustrate the method, the adapted technique of [27], to recover a large majority of the bits, from which the full nonce is then obtained using only one signature. Once the nonce has been fully determined the secret key is obtained using only less than one second of additional computation time (idle computer). For example, we are able to obtain around 546 of the 571 bits of a nonce. Though the goal and approach of the works are similar, the methods are very different.

The attack of [5] uses the above methods to highlight specifically a vulnerability in earlier versions of OpenSSL's ladder implementation for curves over binary fields. Though the attacks differ, they both illustrate that the OpenSSL implementation of the Montgomery ladder is vulnerable from both remote attacks and attacks launched from virtual machines with access to the memory of the target computer. The countermeasure suggested in [5] will not thwart this 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 key and symmetric 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 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 (One word is 16 bits or 32 bits for IA-32 and X86-64 architectures respectively). 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 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 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. While this is a small difference, it is sufficient for mounting an attack that recovers the values of the bits.

Our spy program uses the FLUSH+RELOAD technique to monitor the execution of the `if` statement in line 273. We distinguish between executing the `then` and the `else` blocks of the `if` statement. This information reveals the value of the bit tested by the `if` statement.

FLUSH+RELOAD monitors execution by placing probes on shared memory lines. For the attack to recover the bit value, it must distinguish between memory lines access sequences resulting from a set bit and those resulting from a clear bit. 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 mapping of source lines to cache lines in our build of OpenSSL is depicted in Fig. 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 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 line is accessed when processing of a bit starts. For a set bit, the processing continues with source line 275, which maps to memory

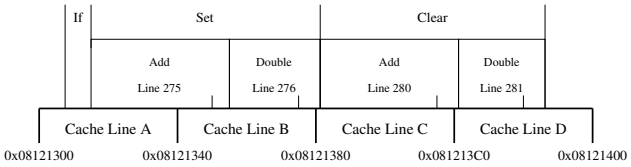

Fig. 1 Mapping from Source Code to Memory

lines *A* and *B*. The actual call to the group add function occurs at address 0x08121347. (See mark in Fig. 1.) 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 2 shows the memory accesses when processing a set bit followed by a clear bit.

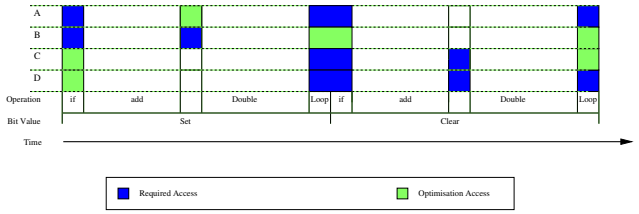

Fig. 2 Observable Memory Access over Time (processing a set then clear bit)

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 [24], 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 [11]. 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 line. 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. 2, 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 access. The first is the attack temporal resolution which affects its ability to accurately capture the sequence of memory access by the victim. The second limitation is the possibility of 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 access 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 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 overlap is lower.

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.

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

4 Experimental Setup and Results

To test the 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 [6] the OpenSSL library. For the experiment we used the OpenSSL `sect1571r1` curve (NIST Binary-Curve B-571 [18].)

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.

Figure 3 shows the results of the probes during 50 time slots. Probes taking less than the threshold of 120 clock cycles indicate a victim access to the probed line. The diagram also indicates the time slots in which the victim program executed the group add and the group double operations.

For example, in the first time slot, the spy program captured access to memory lines *B* and *D*, as well as an access to the last memory line of the group double. The end of the group double is also the end of processing a bit, hence at time slot 1, the victim finished processing a bit and started the next one. The next captured probe is at time slot 5. In this time slot, the victim accessed memory line *B* and started executing a group double. Access to line *B* between the group add and the group double operations indicates that the bit is set. Processing the next bit starts at time slot 8 and ends in time slot 16. The access to memory line *D* in time slot 13 indicates that the second bit is clear.

In the absence access to memory lines *B* or *D* in time slot 42, the captured access to the group double start indicates that the spy process missed a value of a bit. However, the fact that a bit was processed at that time slot is not missed, demonstrating the value of probing the start of the group double operation.

To measure the number of missing bits we traced the computation of 100 signatures. On average, the attack misses only 4.26% of the bits or 25.28 bits per signature. The distribution of number of bits missing per signature is in Fig. 4. The number of missing bits ranges from 10 to 34, with the median at 25 missing bits per signature.

As Fig. 5 demonstrates, the distribution of missing bits position is not uniform. The first bit (bit 0) is always missed. This is mostly the result of the short time it takes OpenSSL to compute the group double operation for the first bit. Even ignoring the first bit, it is evident that missing bits tend to cluster towards the most significant bits of the scalar. Around 15% of bits in positions 1 to 20 are missed, compared with 3.6% of bits from position 50 onward (where the distribution of missing bits is approximately uniform).

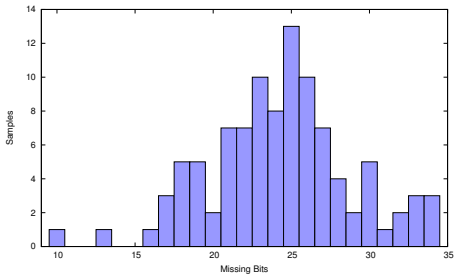

Fig. 4 Missing Bits per Signature

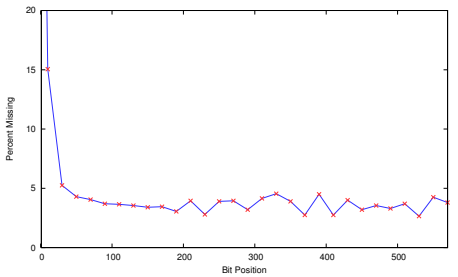

Fig. 5 Missing Bits per Bit Position

5 Discussion

Full Recovery of the Nonce

Given the high proportion of nonce bits recovered by the FLUSH+RELOAD attack, using the LLL based techniques of [10, 19] described in Sect. 2.4 seems computationally excessive. With a worst case of 34 bits missing, the baby steps giant steps algorithm [23] would require less then 10 Megabytes of memory and less than one second of computation to complete the nonce.

Implications

The FLUSH+RELOAD attack is now a broad spectrum attack, it is no longer a "specialised" attack specific to a single implementation. It can be adapted to many ECC protocol implementations on Intel architectures (contrary to comments on [22]) for which the protocol requires a scalar multiplication by a secret entity.

As the ECDLP is not targeted by this attack, the signature protocol is made no more vulnerable by our results. This attack targets the scalar multiplication implementation of OpenSSL and is therefore particular to implementations using this and similar implementations of the Montgomery ladder. The vulnerability introduced by this implementation

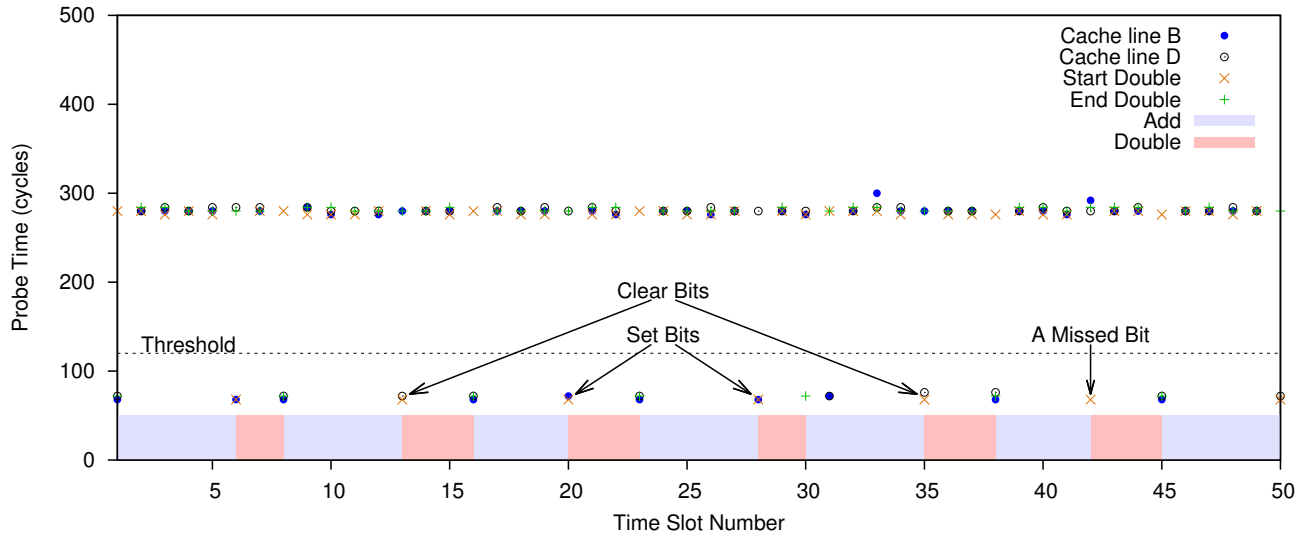

Fig. 3 Probe Timing During Signing

is due to the bits of the secret nonce determinintg which conditional branch is executed. We monitor the branching to determine the value of the bits. This is clearly an attack which needs to be combatted by both software and hardware cooperatively: Our spy program is able to determine how the algorithm executes by having access to the victim's memory and used this knowledge to reconstruct values used by the software.

Mitigation

In the Networking and Cryptography library (NaCl), implemented by Daniel J. Bernstein, Tanja Lange and Peter Schwabe, there is no data flow from secrets to branch conditions, precisely the vulnerability of the OpenSSL implementation targeted by this attack. Analysis of the core security features of NaCl is given in [4]. The attack presented in this article relies on distinguishing bits of the nonce by observing the branching in traditional Montgomery ladder implementation. As the NaCl library avoids branching dependent on secret parameters this attack is not applicable to NaCl's `crypto_sign` API. (NaCl is in the public domain and has been made available by the authors of [4] at `http://nacl.cr.yp.to`.)

6 Conclusions and future work

The results of this work and [5] imply that the OpenSSL montgomery ladder implementation should be avoided in all implementations of elliptic curve protocols when a scalar multiplication step involves a secret parameter. This attack is applicable when the mallicious party has access to the memory of the targeted device, a completely reasonable assumption, possible when using a virtual machine, a computer in multiuser mode, cloud computing or a computer victim to malware.

The results of this work also support the theory of [25] that smaller keys are more resilient to side-channel analysis, in this attack a higer proportion of the nonce was obtained for larger key sizes. This implies that as we naturally transition to larger parameters in response to increasing computing capabilities, prevention of side-channel attacks should be incorporated into the implementation design, as is the methodology adopted by the authors of the NaCl cryptographic library [4].

Future work in this line of research is to adapt this attack to recover more bits when a sliding window is used; using the method presented in this paper we are able to extract the first few bits of the sliding window and thus the proportion of bits obtained decreases as the window size increases. We aim to develop the attack to improve on this proportion.

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.

References

1. Adleman, L.M., Demarrais, J.: A Subexponential Algorithm for Discrete Logarithms over all Finite Fields. Mathematics of computation **61**(203) (1993) 1–15
2. American National Standards Institute: ANSI X9.62, Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm. (1999)
3. Balasubramanian, R., Koblitz, N.: The Improbability that an Elliptic Curve has Subexponential Discrete Log Problem under the Menezes - Okamoto - Vanstone Algorithm. Journal of Cryptology **11**(2) (1998) 141–145

4. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Proceedings of the 2nd international conference on Cryptology and Information Security in Latin America. LATINCRYPT'12, Berlin, Heidelberg, Springer-Verlag (2012) 159–176
5. 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
6. Cipresso, T., Stamp, M.: Software reverse engineering. In Stavroulakis, P., Stamp, M., eds.: Handbook of Information and Communication Security. Springer (2010) 659–696
7. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Improving the parallelized pollard lambda search on anomalous binary curves. Math. Comput. **69**(232) (2000) 1699–1705
8. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory **31**(4) (1985) 469–472
9. Gopalakrishnan, K., Thériault, N., Yao, C.Z.: Solving discrete logarithms from partial knowledge of the key. In Srinathan, K., Pandu Rangan, C., Yung, M., eds.: Progress in Cryptology – INDOCRYPT 2007. Volume 4859 of Lecture Notes in Computer Science., Springer (2007) 224–237
10. Howgrave-Graham, N., Smart, N.P.: Lattice attacks on digital signature schemes. Des. Codes Cryptography **23**(3) (2001) 283–290
11. Intel Corporation: Intel 64 and IA-32 Architecture Optimization Reference Manual. (April 2012)
12. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security **1**(1) (August 2001) 36–63
13. 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
14. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**(177) (January 1987) 203–209
15. Lenstra, A., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. **261** (1982) 515 –534
16. 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 (1985) 417–426
17. Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177) (January 1987) 243–264
18. National Institute of Standards and Technology: FIPS PUB 186-4 Digital Signature Standard (DSS). (2013)
19. 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
20. 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
21. OpenSSL. `http://www.openssl.org`
22. Rauf, A.: Vulnerability note VU#976534 l3 cpu shared cache architecture is susceptible to a Flush+Reload side-channel attack (October 2013)
23. Shanks, D.: Class number, a theory of factorization, and genera. In: Proceedings of the Symposium of Pure Mathematics 20, American Mathematical Society (1971) 415–440
24. 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
25. 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
26. Wiener, M.J., Zuccherato, R.J.: Faster attacks on elliptic curve cryptosystems. In Tavares, S.E., Meijer, H., eds.: Selected Areas in Cryptography. Volume 1556 of Lecture Notes in Computer Science., Springer (1998) 190–200
27. 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/`.