When Constant-time Source Yields Variable-time Binary: Exploiting Curve25519-donna Built with MSVC 2015

Thierry Kaufmann[1], Hervé Pelletier[1], Serge Vaudenay[2], and Karine Villegas[3]

[1] Kudelski Security, Cheseaux, Switzerland,
{thierry.kaufmann, herve.pelletier}@kudelskisecurity.com
[2] EPFL, Lausanne, Switzerland,
serge.vaudenay@epfl.ch
[3] Nagravision, Cheseaux, Switzerland,
karine.villegas@nagra.com

Abstract. The elliptic curve *Curve25519* has been presented as protected against state-of-the-art timing-attacks [2]. This paper shows that a timing attack is still achievable against a particular *X25519* implementation which follows the RFC [4] 7748 requirements [10]. The attack allows the retrieval of the complete private key used in the ECDH protocol. This is achieved due to timing leakage during Montgomery ladder execution and relies on a conditional branch in the Windows runtime library 2015. The attack can be applied remotely.

Keywords: Side-channel, timing attack, ECC, RFC 7748, X25519

1 Introduction

Side-channel attacks are a proven practical means of attack against cryptographic implementations [5]. They make use of physical quantities, e.g., electromagnetic emanations, power consumption, photon emissions, timing variations, etc., to retrieve some sensitive information such as a secret key. Timing attacks were first presented in 1996 by Kocher [8]. They have been shown to be very effective while easily performed. In particular side-channel timing attacks are not intrusive and do not require high-end equipment nor necessarily physical access to the targeted system. Thus they can be applied remotely.

Elliptic curves are increasingly used in cryptography. The asymmetric keys of an ECC implementation are smaller than those required for an RSA implementation with the same cryptographic security. RFC 7748 [10] presents an ECC design that uses regular operations and is thus supposed to be resistant to side-channel timing attacks. The RFC is intended to prevent use of curves which have inherent side-channel leakage weaknesses. Classical side-channel attacks against such poor ECC implementations are published regularly, e.g., [1] and [6].

[4] *Requests for Comments*: document series containing technical and organizational notes about the Internet

In this paper we show that having regular operations is necessary but not sufficient. It is not possible to ensure a side-channel attack proof system in a high level environment. Indeed, we do not have real control over a high performance ecosystem like a server (high-level programming language, compiler/linker, preprocessor options, etc.). In the following sections we present a timing attack capable of retrieving the private key used in the ECDH protocol remotely.

2 State of the Art

In his paper [8], Kocher pointed out that it is common for a cryptosystem to take different amounts of time to execute the same calculation for different inputs. This is due to many factors including code architecture, compiler, processor optimizations, and cache. He showed that rather simple timing attacks could be perpetrated on implementations of Diffie-Hellman, RSA, and DSS. He particularly targeted modular exponentiation, which has a secret or sensitive input-dependent execution.

Later, Brumley and Boneh showed that it was possible to mount remote timing attacks by implementing an attack against OpenSSL [3]. By measuring the time between sending a decryption request to a server and the reception of the response they were able to extract the private key of the server. They exploited the fact that the *sliding window* exponentiation (an optimization of *square and multiply*) uses Montgomery reduction in order to reduce modulo q. This reduction uses an extra step ("extra reduction") in some specific cases and this creates a difference in timing which can be exploited. They showed that the noise due to network communication overhead could be eliminated by sampling several times. Their attacks required however the network to have less than 1ms of variance.

In 2011, Brumley and Tuveri [4] showed that remote timing attacks were still feasible on ECC implementations that were meant to be more resistant to this kind of attack. They showed that the fixed-sequence Montgomery ladder used in the computation of the scalar multiplication was not sufficient to fully protect against their attack. They were able to recover the private key remotely, using a lattice attack [7].

3 Curve25519

The *Curve25519* was first presented by Bernstein in 2006 [2]. It is an elliptic curve of the form $y^2 = x^3 + 486662x^2 + x$, which is birationally equivalent to the Edwards curve: $1 \cdot x^2 + y^2 = 1 + (121665/121666)x^2y^2$. It exists over the field $\mathbb{F}_p$, with $p = 2^{255} - 19$. The order of the base point is the prime $p_1 = 2^{252} + 27742317777372353535851937790883648493$. By design, there is no need for special processing for $\mathcal{O}$ (infinite) or points outside the curve and any 32-byte sequence can be used as public key. Only the x-coordinate of the points is used in the computations.

The implementation of this curve for ECDH (called *X25519*) makes use of a 32-byte secret key and a 32-byte public key. The secret key begins with the bits 01 and the last three bits are set to 0.

The value of the x-coordinate of the base point is 9 with prime order p_1 (written above) over the field $\mathbb{F}_p$.
The exchange works in the following way:

- Each user takes the public string 9 (x-coordinate of the base point on the curve) and multiplies it by their secret key key_i. The result is in fact the public key K_i (only the x-coordinate of the point) of each user.
- The public keys are exchanged and both users then compute $key_i * K_j$, with K_j a point on the curve and key_i the scalar.
- Each user ends up with the point $key_i * key_j * 9$ (respectively $key_j * key_i * 9$), which is the shared secret.

Curve25519 was presented in 2006 by Bernstein with security in mind. This curve naturally provides state-of-the-art timing-attack protection. Particularly the implementation avoids input-dependent branches, input-dependent array indices, and other instructions with input-dependent timings. Moreover, by nature, this curve offers high speed computation and free key compression. In order to speed up the computation the integers are loaded into the floating-point registers. Some parts of the code are directly written in assembly language.

3.1 Curve25519-donna

Adam Langley implemented this curve (in *C*) in order to compute ECDH. This version, called *Donna* [9], follows the RFC recommendations.

It is based on Bernstein's implementation and makes use of a modified version of the Montgomery ladder to compute the point multiplications (for projective coordinates). This Montgomery ladder allows computation of the addition and doubling in an interleaved way and is intended to do this in a constant time regardless of input. It makes use of an accessory swap function whose execution is also expected to execute with constant time. These functions are described in RFC 7748 [10].

The coordinates of the points of the elliptic curve are represented by a reduced-degree reduced-coefficient polynomial and each polynomial coefficient is represented over 64 bits.

Representation of the integers modulo $2^{255}-19$. Elements of $\mathbb{Z}/(2^{255}-19)$ can be seen as elements of R (for $x = 1$), the ring of polynomials $\sum_i u_i x^i$ where u_i is an integer multiple of $2^{\lceil 25.5i \rceil}$.
The coordinates of the points on the curve (integers modulo $2^{255} - 19$) are represented by such a polynomial with the requirement of being reduced-degree and reduced-coefficient.

Reduced-degree means that the degree of the polynomial is small. In this case the maximum degree of the polynomial is 9. Limiting the degree of the

polynomial allows reduction of the number of coefficient multiplications when multiplying the integer.

Reduced-coefficient means restricting the highest possible value of a coefficient. For this implementation, the value of the coefficient $u_i/2^{\lceil 25.5i \rceil}$ is limited from -2^{25} to 2^{25}.

In summary, the coordinates are represented by the 10 coefficients u_0, $u_1/2^{26}$, $u_2/2^{51}$, $u_3/2^{77}$, $u_4/2^{102}$, $u_5/2^{128}$, $u_6/2^{153}$, $u_7/2^{179}$, $u_8/2^{204}$, $u_9/2^{230}$ from the polynomial $u_0 + u_1x + ... + u_9x^9$. The value of a coordinate is given by $X = u_0 + u_1 + ... + u_9$. Note that this representation is not unique, but is faster to compute than the *smallest* representation [2].

4 Attack

4.1 Environment

The attack was performed on two computers running 64-bit Windows 7 OS. The first computer was equipped with the Intel processor *i5-2400* (3.1 GHz, 4 cores, 4 threads, *Sandy Bridge* architecture), while the processor of the second PC was a dual Intel Xeon E5-2630v2 (2.6 GHz, 12 cores, 24 threads, *Ivy Bridge*). The code was compiled for 32-bit architectures using Visual Studio 2015 (MSVC). It makes use of the Windows runtime libraries 2015. The program was written in C.

In order to be as close as possible to a real case we did not change the default enabled options in the BIOS: *Hardware Prefetcher*, *Adjacent Cache Line Prefetcher*, *DCU Streamer Prefetcher*, and *DCU IP Prefetcher*.

Counter. In order to measure timings we made use of the assembly instruction for Intel processors called *rdtsc* (for *Read Time-Stamp Counter*) which allows reading of the time stamp counter of the processor. The time-stamp counter is contained in a 64-bit MSR (Model-Specific Register). Using this command before and after the processing, it was possible to determine the elapsed number of clock cycles. This instruction is not portable as it can only be applied to Intel processors.

4.2 Timing Leakage Observation

Although the computation of the scalar multiplication should be time-constant, we spotted some timing differences depending on the value of the key. When comparing the mean value (over 10,000 measurements) of the execution times of the multiplication of the base point 9 with either the same key or different keys, we observed some differences (Figure 1). We made three observations from the results shown on Figure 1.

First, the counter induces some timing differences. Two identical executions do not take the same number of cycles.

Second, the variance of the timing executions is different for the execution of

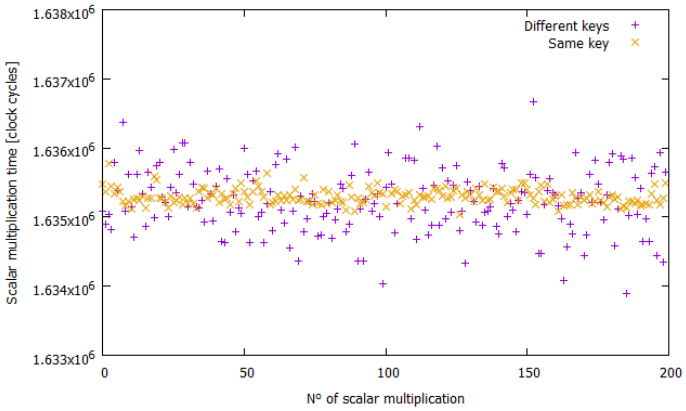

Fig. 1. Computation times depending on the key.

the same computation and the computation with different values of key. This implies some input-dependent instructions.
Third, the timing difference seen is small. For 256 bits the difference is at most 4000 clock cycles while the overall computation takes about 1.6 million clock cycles.

4.3 Timing Leakage Origin

After a careful analysis of the binary code, the observed timing leakage appeared to be coming from the assembly function *llmul.asm* found in the Windows runtime library[5]. The function *llmul.asm* is called to compute the multiplication of two 64-bit integers. It contains a branch condition which causes differences in execution time (see Figure 2, line 65). If both operands of the multiplication have their 32 most significant bits equal to 0 then the multiplication of these words is avoided as the computation is correctly judged to be 0.

This runtime function is called by the program when executing the multiplication with a constant *fscalar_product* (Listing 1.1) in the Montgomery ladder algorithm described in [10]. There was no way to see that this ordinary multiplication could cause a difference of timing, especially as a carry is never needed with coefficients being smaller than 2^{25}.

Listing 1.1. Code in fscalar_product function

```
for (i = 0; i < 10; ++i) {
        output[i] = in[i] * scalar;
}
```

We saw in Section 3.1 that the coefficients representing the coordinates are bounded between -2^{25} and 2^{25}, thus smaller than 2^{32} and representable over

[5] A runtime library is a library for a specific environment. It contains pieces of code that can be called by a program executed in this environment.

```
61          mov     eax,HIWORD(A)
62          mov     ecx,HIWORD(B)
63          or      ecx,eax         ;test for both hiwords zero.
64          mov     ecx,LOWORD(B)
65          jnz     short hard      ;both are zero, just mult ALO and BLO
66          mov     eax,LOWORD(A)
67          mul     ecx
68          ret     16              ; callee restores the stack
```

Fig. 2. Part of the code of the Microsoft *llmul* function with incriminating line

32 bits only. However, two's complement representation is used and negative numbers are represented with leading ones. As the choice was made to work with 64-bit integers, for negative numbers the 32 most significant bits are all ones.

In addition, the constant *a24* (121665) is a positive integer represented over 32 bits and the 32 most significant bits are therefore all 0. Thus the execution of the second part of the code of *llmul* only depends on the sign of the coefficient. We can say that the computation time of a key bit (k_i) in the Montgomery ladder is dependent on the number of negative coefficients representing the coordinates of the point being processed.

4.4 Timing Attack

In the Montgomery ladder, the value of the key bit k_i only decides which coordinates will be doubled and which ones will be added. Let's call c_{ij} the j^{th} coefficient of the polynomial representing the intermediate value of the new Z-coordinate of the point which is multiplied by the scalar for the bit k_i in the Montgomery ladder.

The values of the coefficients c_{ij}'s depend on several parameters: the base point, the values of the previously processed bits of the key, and k_i. For a given base point and fixing the previous bits (more significant) it is possible to count the number of negative coefficients among the c_{ij}'s, by executing the code until k_i. Depending on the base point and the key there can be a different number of negative coefficients when $k_i = 0$ or $k_i = 1$. This difference can go from 0 to 10 which is still a very small difference in terms of clock cycles.

Attack core idea. We can see the overall computation time (called F below) as the time required to process all the previous bits, the attacked bit and the next bits. For a key $k = k_{l-1}k_{l-2}...k_1k_0$ and a base point P, we have:

$$\begin{aligned} F(k,P) = &\sum_{j>i} f(k_j|k_{l-1},k_{l-2},...,k_{j+1},P) \\ &+ f(k_i|k_{l-1},k_{l-2},...,k_{i+1},P) \\ &+ \sum_{j<i} f(k_j|k_{l-1},k_{l-2},...,k_i,...,k_{j+1},P) \end{aligned} \tag{1}$$

where f is the time of the processing of 1 bit in the Montgomery ladder.

Keeping this in mind, if we choose base points so that there is the same number of negative coefficients for the i$^{\text{th}}$ bit k_i the time due to the processing of the other bits (before and after) can be assumed to be random from one execution to another.
Thus taking the mean over n executions with different base points P_j's, we have:

$$F_\mu(k, P_j\text{'s}) = \frac{1}{n}\sum_{j=1}^{n}\left(f(k_i|k_{l-1}, k_{l-2}, ..., k_{i+1}, P_j) + N(\mu_N, \sigma^2)\right) \tag{2}$$

where N is some Gaussian noise of mean μ_N and standard deviation σ. We know that the average of Gaussian noises tends to the mean:

$$F_\mu(k, P_j\text{'s}) \cong \frac{1}{n}\sum_{j=1}^{n} f(k_i|k_{l-1}, k_{l-2}, ..., k_{i+1}, P_j) + \mu_N \tag{3}$$

Then, for different sets of base points A and B:

$$\begin{gathered} F_\mu(k, P_j\text{'s} \in A) > F_\mu(k, P_j\text{'s} \in B) \Rightarrow \\ \frac{1}{n}\sum_{j=1}^{n} f(k_i|k_{l-1}, ..., k_{i+1}, P_j\text{'s} \in A) > \frac{1}{n}\sum_{j=1}^{n} f(k_i|k_{l-1}, ..., k_{i+1}, P_j\text{'s} \in B) \end{gathered} \tag{4}$$

If we select base points causing more negative coefficients when the bit k_i is 0 or 1 respectively (let's call the sets $high_0$ and $high_1$ respectively) and compare the overall computation times, we are able to find the value of k_i:

$$k_i = \begin{cases} 0, & \text{if } F_\mu(k, Pi_j\text{'s} \in high_0) > F_\mu(k, Pi_j\text{'s} \in high_1) \\ 1, & \text{otherwise} \end{cases} \tag{5}$$

Timing measurements. The attack procedure is described in Algorithm 1. We maintain a *constructed* key (key_c) with the bits we found from the *unknown* key (key_u). The base points are chosen by picking a point value at random, executing the scalar multiplication routing and simply counting the number of negative coefficients for the cases when $k_i = 0$ and $k_i = 1$. We want the difference between those two values to be at least 8. If we compare the average of the times to compute the scalar multiplication for base points of $high_0$ with the mean of the times for base points of $high_1$ then the difference does not depend on the rest of the bits.

It can be noted that the difference in the number of negative coefficients between base points of $high_0$ and the ones of the base points of $high_1$ is at least 6 (a high value is at least 8 and a low value can be at most 2, thus the minimum is $8 - 2 = 6$).

Algorithm 1 Attack Procedure

1: Knowing the i first bits of key_u, set the first bits of key_c to these values
2: Executing the code separately with key_c and a random base point, we count the number of negative coefficients in the polynomial representation of the point when multiplying by the bit $i+1$, for $k_{i+1}=0$ and $k_{i+1}=1$. We call them $coeff_0$ and $coeff_1$ respectively.
3: **if** $coeff_0 - coeff_1 > 7$ **then**
4: Add base point to the set $high_0$
5: **else**
6: **if** $coeff_0 - coeff_1 < -7$ **then**
7: Add base point to the set $high_1$
8: Repeat steps 2 to 7 until we have 200 points in each set.
9: For each base point $high_0$ and $high_1$, compute 25,000 times the scalar multiplication with key_u and measure the overall time of execution.
10: For each base point, take the mean value over the 15 minimum values of timing measured. We call these means μ_i's.
11: Compute the mean (μ_{high_0}) of the μ_i's for base points in $high_0$
12: Compute the mean (μ_{high_1}) of the μ_i's for base points in $high_1$
13: If $\mu_{high_0} > \mu_{high_1}$, then $k_{i+1}=0$, otherwise $k_{i+1}=1$
14: Repeat steps 2 to 13 until the same value for k_{i+1} was found twice.
15: Set k_{i+1} to the value found twice and go to 1 for the next bit

5 Results

Although the distribution of the μ_i's is not a clean Gaussian distribution it is possible to get some coherent results when taking the minimum values. When comparing the average values of the $high_0$ and $high_1$ base points computation times, we observe a difference as expected (Figure 3). In order to obtain some more accurate timing measurements, each ECC execution was associated to a specific core (affinity selection) on the server. It seems that, contrary to Linux systems, Windows has an aggressive management of power consumption of cores, which induces bigger variations in the timing measurements.

5.1 Evaluation of the Attack

This attack works but it takes some time to recover all the bits of the key. The time required to compute one ECC computation is around 1.6 million clock cycles. 25,000 measurements per base point are needed. There are 400 base points per bit and this is performed 2 to 3 times per bit. On the Intel Xeon processor, at 2.6 GHz, this represents about 15 seconds for the 25,000 measurements. As 5 bits are fixed we need to recover 251 bits. Hence, we need about 1 month to recover the whole key with this method. It is probable that an optimization of the number of measurements and base points would decrease the time needed.

The measurements need to be repeated many times because the execution times can vary significantly and we are looking for a very small difference in timing. Taking the mean over the 15 minimum values appears to be a good

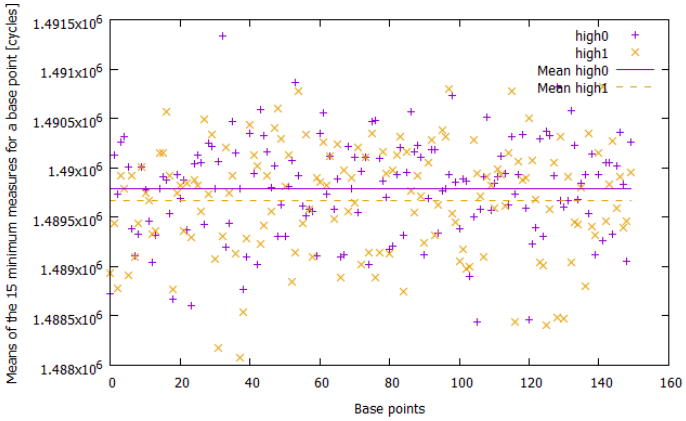

Fig. 3. Comparison of the means of $high_0$ and $high_1$ for $k_i = 0$

compromise as the minimum value is more stable than the mean of the measures but, if this minimum value is rare (e.g., because of what is running on the same processor), it can lead to errors. This is certainly not the most efficient way. It is however robust and allows targeting of all sorts of systems which may have very different behaviors. Creating a precise model of the distribution of the measurements would take time and would remain specific to a given system.

As the attack requires knowledge of the bits preceding the bit being attacked it is not possible to target a specific bit of the key without processing all the previous bits. Furthermore, once a mistake is made in the presumed value of a bit, the chance of correctly recovering the subsequent bits is negligible (the attack makes no sense as we choose the base points $high_0$ and $high_1$ for another key).

Fine tuning the number of measurements and base points could be performed in order to decrease the time of the attack. Furthermore, when there are only a few bits left to recover, a brute-force attack might be faster than continuing the attack until the very last bit of the key.

Other analyses based on the "profiled attack" approach with a parametric template (on a multi-dimensional Gaussian model) or a machine learning system could be interesting to investigate.

5.2 Extension to Remote Attacks.

As the overall times of computation are measured the attack was expected to also be feasible remotely. In order to test this hypothesis we measured response times of network communications on a local server (ping requests). When we added the network delays to the timings of the overall computations and applied the attack we were still able to retrieve the correct values of the bits.

6 Conclusion

It has been shown above that simply following the recommendations of the RFC and having a "constant-time" source code is not sufficient to prevent timing leakage. Once a security design is implemented, whatever effort is put into protecting each part of the code, there still remains a strong possibility of a timing leak. It is virtually *impossible* to have control over all the parameters at stake. Compiler and processor optimizations, processor specificities, hardware construction, and runtime libraries are all examples of elements that cannot be predicted when implementing at a high level.

The attack developed shows that the effects of these low-level actors can be exploited practically for the curve X25519. It is not only theoretically possible to find weaknesses, they can be found and exploited in a reasonable amount of time.

Nevertheless, the idea of ensuring that the design itself is secure by using a formalized approach such as RFC is however an important step in minimizing the side-channel leakage of any final system.

This paper also highlights one particular aspect: the potential weakness of other codes implemented using the Windows runtime library.

References

1. Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi: Side-Channel Analysis of Weierstrass and Koblitz Curve ECDSA on Android Smartphones. Cryptology ePrint Archive, Report 2016/231 (2016)
2. Daniel J. Bernstein: Curve25519: new Diffie-Hellman speed records. In *9th international conference on theory and practice in public key cryptography*, 207–228 (2006)
3. David Brumley and Dan Boneh: Remote timing attacks are practical. *Computer Networks*, 701–716 (2005)
4. Billy Bob Brumley and Nicolas Tuveri: Remote timing attacks are still practical. *ESORICS* (2011)
5. Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi: Template attacks. In *CHES 2002 Proceedings from Springer-Verlag*, 13–28 (2002)
6. Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom: ECDSA key extraction from mobile devices via nonintrusive physical side channels. Cryptology ePrint Archive, Report 2016/230 (2016)
7. Nick Howgrave-Graham and Nigel P. Smart: Lattice attacks on digital signature schemes. *Des. Codes Cryptography* 23(3):283–290 (2001).
8. P. Kocher: Timing attack on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology: Proceedings of CRYPTO '96*, 104–113. Springer-Verlag, (1996)
9. Adam Langley: Implementation of curve25519-donna. http://code.google.com/p/curve25519-donna. Accessed: 2015-09-16
10. Sean Turner, Adam Langley, and Mike Hamburg: Elliptic Curves for Security. IETF RFC 7748 (January 2016)