"Ooh Aah... Just a Little Bit" : A Small Amount of Side Channel Can Go a Long Way

Naomi Benger[1], Joop van de Pol[2], Nigel P. Smart[2], and Yuval Yarom[1]

[1] School of Computer Science, The University of Adelaide, Australia
mail.for.minnie@gmail.com, yval@cs.adelaide.edu.au
[2] Dept. Computer Science, University of Bristol, United Kingdom
joop.vandepol@bristol.ac.uk, nigel@cs.bris.ac.uk

Abstract. We apply the FLUSH+RELOAD side-channel attack based on cache hits/misses to extract a small amount of data from OpenSSL ECDSA signature requests. We then apply a "standard" lattice technique to extract the private key, but unlike previous attacks we are able to make use of the side-channel information from almost all of the observed executions. This means we obtain private key recovery by observing a relatively small number of executions, and by expending a relatively small amount of post-processing via lattice reduction. We demonstrate our analysis via experiments using the curve **secp256k1** used in the Bitcoin protocol. In particular we show that with as little as 200 signatures we are able to achieve a reasonable level of success in recovering the secret key for a 256-bit curve. This is significantly better than prior methods of applying lattice reduction techniques to similar side channel information.

1 Introduction

One important task of cryptographic research is to analyze cryptographic implementations for potential security flaws. This aspect has a long tradition, and the most well known of this line of research has been the understanding of side-channels obtained by power analysis, which followed from the initial work of Kocher and others [22]. More recently work in this area has shifted to looking at side-channels in software implementations, the most successful of which has been the exploitation of cache-timing attacks, introduced in 2002 [32]. In this work we examine the use of spy-processes on the OpenSSL implementation of the ECDSA algorithm.

OpenSSL [31] is an open source tool kit for the implementation of cryptographic protocols. The library of functions, implemented using C, is often used for the implementation of Secure Sockets Layer and Transport Layer Security protocols and has also been used to implement OpenPGP and other cryptographic standards. The library includes cryptographic functions for use in Elliptic Curve Cryptography (ECC), and in particular ECDSA. In particular we will examine the application of the FLUSH+RELOAD attack, first proposed by Yarom and Falkner [40], then adapted to the case of OpenSSL's implementation of ECDSA over binary fields by Yarom and Benger [39], running on X86 processor architecture. We exploit a property of the Intel implementation of the X86 and X86_64 processor architectures using the FLUSH+RELOAD cache side-channel attack [39, 40] to partially recover the ephemeral key used in ECDSA.

L. Batina and M. Robshaw (Eds.): CHES 2014, LNCS 8731, pp. 75–92, 2014.

In Yarom and Benger [39] the case of characteristic two fields was considered, but the algorithms used by OpenSSL in the characteristic two and prime characteristic cases are very different. In particular for the case of prime fields one needs to perform a post-processing of the side-channel information using cryptanalysis of lattices. We adopt a standard technique [21, 30] to perform this last step, but in a manner which enables us to recover the underlying secret with few protocol execution runs. This is achieved by using as much information obtained in the FLUSH+RELOAD step as possible in the subsequent lattice step.

We illustrate the effectiveness of the attack by recovering the secret key with a very high probability using only a small number of signatures. After this, we are able to forge unlimited signatures under the hidden secret key. The results of this attack are not limited to ECDSA but have implications for many other cryptographic protocols implemented using OpenSSL for which the scalar multiplication is performed using a sliding window and the scalar is intended to remain secret.

Related Work. Microarchitectural side-channel attacks have been used against a number of implementations of cryptosystems. These attacks often target the L1 cache level [1, 2, 5, 10, 13, 14, 37, 41] or the branch prediction buffer [3, 4]. The use of these components is limited to a single execution core. Consequently, the spy program and the victim must execute on the same execution core of the processor. Unlike these attacks, the FLUSH+RELOAD attack we use targets the last level cache (LLC). As the LLC is shared between cores, the attack can be mounted between different cores.

The attack used by Gullasch et al. [20] against AES, is very similar to FLUSH+RELOAD. The attack, however, requires the interleaving of spy and victim execution on the same processor core, which is achieved by relying on a scheduler bug to interrupt the victim and gain control of the core on which it executes. Furthermore, the Gullasch et al. attack results in a large number of false positives, requiring the use of a neural network to filter the results.

In [40], Yarom and Falkner first describe the FLUSH+RELOAD attack and use it to snoop on the square-and-multiply exponentiation in the GnuPG implementation of RSA and thus retrieve the RSA secret key from the GnuPG decryption step. The OpenSSL (characteristic two) implementation of ECDSA was also shown to be vulnerable to the FLUSH+RELOAD attack [39]; around 95% of the ephemeral private key was recovered when the Montgomery ladder was used for the scalar multiplication step. The full ephemeral private key was then recovered at very small cost using a Baby-Step-Giant-Step (BSGS) algorithm. Knowledge of the ephemeral private key leads to recovery of the signer's private key, thus fully breaking the ECDSA implementation using only one signature.

One issue hindering the extension of the attack to implementations using the sliding window method for scalar multiplications instead of the Montgomery ladder is that only a lower proportion of the bits of the ephemeral private key can be recovered so the BSGS reconstruction becomes infeasible. It is to extend the FLUSH+RELOAD attack to implementations which use sliding window exponentiation methods that this paper is addressed.

Suppose we take a single ECDLP instance, and we have obtained partial information about the discrete logarithm. In [19, 26, 36] techniques are presented which reduce the search space for the underlying discrete logarithm when various types of partial information is revealed. These methods work quite well when the information leaked is considerable for the single discrete logarithm instance; as for example evidenced by the side-channel attack of [39] on the Montgomery ladder. However, in our situation a different approach needs to be taken.

Similar to several past works, e.g. [10, 11, 27], we will exploit a well known property of ECDSA, that if a small amount of information about each ephemeral key in each signature leaks, for a number of signatures, then one can recover the underlying secret using a lattice based attack [21, 30]. The key question arises as to how many signatures are needed so as to be able to extract the necessary side channel information to enable the lattice based attack to work. The lattice attack works by constructing a lattice problem from the obtained digital signatures and side channel information, and then applying lattice reduction techniques such as LLL [23] or BKZ [35] to solve the lattice problem. Using this methodology Nguyen and Shparlinski [30], suggest that for an elliptic curve group of order around 160 bits, their probabilistic algorithm would obtain the secret key using an expected 23×2^7 signatures (assuming independent and uniformly at random selected messages) in polynomial time, using only seven consecutive least significant leaked bits of each ephemeral private key. A major issue of their attack in practice is that it seems hard to apply when only a few bits of the underlying ephemeral private key are determined.

Our Contribution. Through the FLUSH+RELOAD attack we are able to obtain a significant proportion of the ephemeral private key bit values, but they are not clustered but in positions spread through the length of the ephemeral private key. As a result, we only obtain for each signature a few (maybe only one) consecutive bits of the ECDSA ephemeral private key, and so the technique described in [30] does not appear at first sight to be instantly applicable. The main contribution of this work is to combine and adapt the FLUSH+RELOAD attack and the lattice techniques. The FLUSH+RELOAD attack is refined to optimise the proportion of information which can be obtained, then the lattice techniques are adapted to utilize the information in the acquired data in an optimal manner. The result is that we are able to reconstruct secret keys for 256 bit elliptic curves with high probability, and low work effort, after obtaining less than 256 signatures.

We illustrate the effectiveness of the attack by applying it to the OpenSSL implementation of ECDSA using a sliding window to compute scalar multiplication, recovering the victims's secret key for the elliptic curve **secp256k1** used in Bitcoin [28]. The implementation of the **secp256k1** curve in OpenSSL is interesting as it uses the wNAF method for exponentiation, as opposed to the GLV method [18], for which the curve was created. It would be an interesting research topic to see how to apply the FLUSH+RELOAD technique to an implementation which uses the GLV point multiplication method.

In terms of the application to Bitcoin an obvious mitigation against the attack is to limit the number of times a private key is used within the Bitcoin protocol. Each wallet

corresponds to a public/private key pair, so this essentially limits the number of times one can spend from a given wallet. Thus, by creating a chain of wallets and transferring Bitcoins from one wallet to the next it is easy to limit the number of signing operations carried out by a single private key. See [9] for a discussion on the distribution of public keys currently used in the Bitcoin network.

The remainder of the paper is organised as follows: In 2 we present the background on ECDSA and the signed sliding window method (or wNAF representation) needed to understand our attack. Then in 3 we present our methodology for applying the FLUSH+ RELOAD attack on the OpenSSL implementation of the signed sliding window method of exponentiation. Then in 4 we use the information so obtained to create a lattice problem, and we demonstrate the success probability of our attack.

2 Mathematical Background

In this section we present the mathematical background to our work, by presenting the wNAF/signed window method of point multiplication which is used by OpenSSL to implement ECDSA in the case of curves defined over prime finite fields.

Scalar Multiplication Using wNAF. In OpenSSL the scalar multiplication in the signing algorithm is implemented using the wNAF algorithm. Suppose we wish to compute $[d]P$ for some integer value $d \in [0,\ldots,2^{\ell}]$, the wNAF method utilizes a small amount of pre-processing on P and the fact that addition and subtraction in the elliptic curve group have the same cost, so as to obtain a large performance improvement on the basic binary method of point multiplication. To define wNAF a window size w is first chosen, which for OpenSSL, and the curve **secp256k1**, we have $w = 3$. Then $2^w - 2$ extra points are stored, with a precomputation cost of $2^{w-1} - 1$ point additions, and one point doubling. The values stored are the points $\{\pm G, \pm[3]G, \ldots, \pm[2^w - 1]G\}$.

The next task is to convert the integer d into so called Non-Adjacent From (NAF). This is done by the method in Algorithm 1 which rewrites the integer d as a sum $d = \sum_{i=0}^{\ell-1} d_i \cdot 2^i$, where $d_i \in \{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$. The Non-Adjacent From is so named as for any d written in NAF, the output values $d_0, \ldots, d_{\ell-1}$, are such that for every non-zero element d_i there are at least $w+1$ following zero values.

Once the integer d has been re-coded into wNAF form, the point multiplication can be carried out by Algorithm 2. The occurrence of a non-zero d_i controls when an addition is performed, with the precise value of d_i determining which point from the list is added.

Before ending this section we note some aspects of the algorithm, and how these are exploited in our attack. A spy process, by monitoring the cache hits/misses, can determine when the code inside the **if–then** block in Algorithm 2 is performed. This happens when the element d_i is non-zero, which reveals the fact that the following $w+1$ values $d_{i+1}, \ldots, d_{i+w+1}$ are all zero. This reveals some information about the value d, but not enough to recover the value of d itself.

Instead we focus on the last values of d_i processed by Algorithm 2. We can determine precisely how many least significant bits of d are zero, which means we can determine at least one bit of d, and with probability $1/2$ we determine two bits, with probability

Input: scalar d and window width w
Output: d in wNAF: $d_0, \ldots, d_{\ell-1}$
$\ell \leftarrow 0$
while $d > 0$ **do**
  if $d \bmod 2 = 1$ **then**
    $d_\ell \leftarrow d \bmod 2^{w+1}$
    if $d_\ell \geq 2^w$ **then**
      $d_\ell \leftarrow d_\ell - 2^{w+1}$
    end
    $d = d - d_\ell$
  else
    $d_\ell = 0$
  end
  $d = d/2$
  $\ell += 1$
end

Algorithm 1. Conversion to Non-Adjacent Form

Input: scalar d in wNAF $d_0, \ldots, d_{\ell-1}$ and precomputed points $\{G, \pm[3]G, \pm[5]G, \ldots, \pm[2^w - 1]G\}$
Output: $[d]G$
$Q \leftarrow \mathcal{I}$
for j *from* $\ell - 1$ *downto* 0 **do**
  $Q \leftarrow [2]Q$
  if $d_j \neq 0$ **then**
    $Q \leftarrow Q + [d_j]G$
  end
end

Algorithm 2. Computation of kG using OpenSSL wNAF

$1/4$ we determine three bits and so on. Thus we not only extract information about whether the least significant bits are zero, but we also use the information obtained from the first non-zero bit.

In practice in the OpenSSL code the execution of scalar multiplication by the ephemeral key k is slightly modified. Instead of computing $[k]G$, the code computes $[k + \lambda \cdot n]G$ where $\lambda \in \{1,2\}$ is chosen such that $\lfloor \log_2(k + \lambda \cdot n) \rfloor = \lfloor \log_2(n) \rfloor + 1$. The fixed size scalar provides protection against the Brumley and Tuveri remote timing attack [11]. For the **secp256k1** curve, n is $2^{256} - \varepsilon$ where $\varepsilon < 2^{129}$. The case $\lambda = 2$, therefore, only occurs for $k < \varepsilon$. As the probability of this case is less than 2^{-125}, we can assume the wNAF algorithm is applied with $d = k + n$.

3 Attacking OpenSSL

In prior work the Montgomery ladder method of point multiplication was shown to be vulnerable to a FLUSH+RELOAD attack [39]. This section discusses the wNAF im-

plementation of OpenSSL and demonstrates that it is also vulnerable. Unlike the side-channel in the Montgomery ladder implementation, which recovers enough bits to allow a direct recovery of the ephemeral private key [39], the side-channel in the wNAF implementation only leaks an average of two bits in each window. Consequently, a further algebraic attack is required to recover the private key. This section describes the FLUSH+RELOAD attack, and its application to the OpenSSL wNAF implementation. The next section completes the recovery of the secret key.

FLUSH+RELOAD is a cache side-channel attack that exploits a property of the Intel implementation of the X86 and X86_64 processor architectures, which allows processes to manipulate the cache of other processes [39, 40].

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. Furthermore, pages shared through the use of memory de-duplication in virtualized environments [6, 38] are also susceptible and using them the attack can be applied between co-located virtual machines.

The spy program needs to execute on the same physical processor as the victim, however, unlike most cache-based side channel attacks, our spy monitors access to the last-level cache (LLC). As the LLC is shared between the processing cores of the processor, the spy does not need to execute on the same processing core as the victim. Consequently, the attack is applicable to multi-core processors and is not dependent on hyperthreading or on exploitable scheduler limitations like other published microarchitectural side-channel attacks.

```
Input: adrs—the probed address
Output: true if the address was accessed by the victim
begin
    evict(adrs)
    wait_a_bit()
    time ← current_time()
    tmp ← read(adrs)
    readTime ← current_time()-time
    return readTime < threshold
end
```

Algorithm 3. FLUSH+RELOAD Algorithm

To monitor access to memory, the spy repeatedly evicts the contents of the monitored memory from the LLC, waits for some time and then measures the time to read the contents of the monitored memory. See Algorithm 3 for a pseudo-code of the attack. As reading from the LLC is much faster than reading from memory, the spy can differentiate between these two cases. If, following the wait, the contents of the memory is retrieved from the cache, it indicates that another process has accessed the memory. Thus, by measuring the time to read the contents of the memory, the spy can decide whether the victim has accessed the monitored memory since the last time it was evicted.

Monitoring access to specific memory lines is one of the strengths of the FLUSH+RELOAD technique. Other cache-based tracing techniques monitor access to sets of memory lines that map to the same cache set. The use of specific memory lines reduces the chance of false positives. Capturing the access to the memory line, therefore, indicates that the victim executes and has accessed the line. Consequently, FLUSH+RELOAD does not require any external mechanism to synchronize with the victim.

We tested the attack on an HP Elite 8300 running Fedora 18. The machine features an Intel Core i5-3470 processor, with four execution cores and a 6MB LLC. As the OpenSSL package shipped with Fedora does not support ECC, we used our own build of OpenSSL 1.0.1e. For the experiment we used the curve **secp256k1** which is used by Bitcoin.

For the attack, we used the implementation of FLUSH+RELOAD from [40]. The spy program divides time into time slots of approximately 3,000 cycles (almost 1μs). In each time slot the spy probes memory lines in the group add and double functions. (`ec_GFp_simple_add` and `ec_GFp_simple_dbl`, respectively.) The time slot length is chosen to ensure that there is an empty slot during the execution of each group operation. This allows the spy to correctly distinguish consecutive doubles.

The probes are placed on memory lines which contain calls to the field multiplication function. Memory lines containing call sites are accessed both when the function is called and when it returns. Hence, by probing these memory lines, we reduce the chance of missing accesses due to overlaps with the probes. See [40] for a discussion of overlaps.

To find the memory lines containing the call sites we built OpenSSL with debugging symbols. These symbols are not loaded at run time and do not affect the performance of the code. The debugging symbols are, typically, not available for attackers, however their absence would not present a major obstacle to a determined attacker who could use reverse engineering [16].

4 Lattice Attack Details

We applied the above process on the OpenSSL implementation of ECDSA for the curve **secp256k1**. We fixed a public key $Q = [\alpha]G$, and then monitored via the FLUSH+RELOAD spy process the generation of a set of d signature pairs (r_i, s_i) for $i = 1, \ldots, d$. For each signature pair there is a known hashed message value h_i and an unknown ephemeral private key value k_i.

Using the FLUSH+RELOAD side-channel we also obtained, with very high probability, the sequence of point additions and doubling used when OpenSSL executes the operation $[k_i + n]G$. In particular, this means we learn values c_i and l_i such that

$$k_i + n \equiv c_i \pmod{2^{l_i}},$$

or equivalently

$$k_i \equiv c_i - n \pmod{2^{l_i}}.$$

Where l_i denotes the number of known bits. We can also determine the length of the known run of zeroes in the least significant bits of $k_i + n$, which we will call z_i. In

presenting the analysis we assume the d signatures have been selected such that we already know that the value of $k_i + n$ is divisible by 2^Z, for some value of Z, i.e. we pick signatures for which $z_i \geq Z$. In practice this means that to obtain d such signatures we need to collect (on average) $d \cdot 2^Z$ signatures in total.

We write $a_i = c_i - n \pmod{2^{l_i}}$. For example, writing A for an add, D for a double and X for a *don't know*, we can read off c_i, l_i and z_i from the least execution sequence obtained in the FLUSH+RELOAD analysis. In practice the FLUSH+RELOAD attack is so efficient that we are able to identify A's and D's with almost 100% certainty, with only $\varepsilon = 0.55\% - 0.65\%$ of the symbols turning out to be *don't know*s. To read off the values we use the following table (and its obvious extension), where we present the approximate probability of our attack revealing this sequence.

Sequence	c_i	l_i	z_i	Pr $\approx$
$\ldots X$	0	0.0	0	ε
$\ldots A$	1	1.0	0	$(1-\varepsilon)/2$
$\ldots XD$	0	1.0	1	$\varepsilon \cdot (1-\varepsilon)/2$
$\ldots AD$	2	2.0	1	$((1-\varepsilon)/2)^2$
$\ldots XDD$	0	2.0	2	$\varepsilon \cdot ((1-\varepsilon)/2)^2$
$\ldots ADD$	4	3.0	2	$((1-\varepsilon)/2)^3$

For a given execution of the FLUSH+RELOAD attack, from the table we can determine c_i and l_i, and hence a_i. Then, using the standard analysis from [29, 30], we determine the following values

$$t_i = \lfloor r_i/(2^{l_i} \cdot s_i) \rfloor_n,$$
$$u_i = \lfloor (a_i - h_i/s_i)/2^{l_i} \rfloor_n + n/2^{l_i+1},$$

where $\lfloor \cdot \rfloor_n$ denotes reduction modulo n into the range $[0, \ldots, n)$. We then have that

$$v_i = |\alpha \cdot t_i - u_i|_n < n/2^{l_i+1}, \tag{1}$$

where $|\cdot|_n$ denotes reduction by n, but into the range $(-n/2, \ldots, n/2)$. It is this latter equation which we exploit, via lattice basis reduction, so as to recover d. The key observation found in [29, 30] is that the value v_i is smaller (by a factor of 2^{l_i+1}) than a random integer. Unlike prior work in this area we do not (necessarily) need to just select those executions which give us a "large" value of z_i, say $z_i \geq 3$. Prior work fixes a minimum value of z_i (or essentially equivalently l_i) and utilizes this single value in all equations such as (1). If we do this we would need to throw away all bar $1/2^{z_i+1}$ of the executions obtained. By maintaining full generality, i.e. a variable value of z_i (subject to the constraint $z_i \geq Z$) in each instance of (1), we are able to utilize all information at our disposal and recover the secret key α with very little effort indeed.

The next task is to turn the equations from (1) into a lattice problem. Following [29, 30] we do this in one of two possible ways, which we now recap on.

Attack via CVP: We first consider the lattice $L(B)$ in $d+1$-dimensional real space, generated by the rows of the following matrix

$$B = \begin{pmatrix} 2^{l_1+1} \cdot n & & & \\ & \ddots & & \\ & & 2^{l_d+1} \cdot n & \\ 2^{l_1+1} \cdot t_1 & \dots & 2^{l_d+1} \cdot t_d & 1 \end{pmatrix}.$$

From (1) we find that there are integers $(\lambda_1, \dots, \lambda_d)$ such that if we set $\mathbf{x} = (\lambda_1, \dots, \lambda_d, \alpha)$ and $\mathbf{y} = (2^{l_1+1} \cdot v_1, \dots, 2^{l_d+1} \cdot v_d, \alpha)$ and $\mathbf{u} = (2^{l_1+1} \cdot u_1, \dots, 2^{l_d+1} \cdot u_d, 0)$, then we have

$$\mathbf{x} \cdot B - \mathbf{u} = \mathbf{y}.$$

We note that the 2-norm of the vector $\mathbf{y}$ is about $\sqrt{d+1} \cdot n$, whereas the lattice determinant of $L(B)$ is $2^{d+\Sigma l_i} \cdot n^d$. Thus the vector $\mathbf{u}$ is a close vector to the lattice. Solving the Closest Vector Problem (CVP) with input B and $\mathbf{u}$ therefore reveals $\mathbf{x}$ and hence the secret key α.

Attack via SVP: It is often more effective in practice to solve the above CVP problem via the means of embedding the CVP into a Shortest Vector Problem (SVP) in a slightly bigger lattice. In particular we take the lattice $L(B')$ in $d+2$-dimensional real space generated by the rows of the matrix

$$B' = \begin{pmatrix} B & 0 \\ \mathbf{u} & n \end{pmatrix}.$$

This lattice has determinant $2^{d+\Sigma l_i} \cdot n^{(d+1)}$, by taking the lattice vector generated by $\mathbf{x}' = (\mathbf{x}, \alpha, -1)$ we obtain the lattice vector $\mathbf{y}' = \mathbf{x}' \cdot B' = (\mathbf{y}, -n)$. The 2-norm of this lattice vector is roughly $\sqrt{d+2} \cdot n$. We expect the second vector in a reduced basis to be of size $c \cdot n$, and so there is a "good" chance for a suitably strong lattice reduction to obtain a lattice basis whose second vector is equal to $\mathbf{y}'$. Note, the first basis vector is likely to be given by $(-t_1, \dots, -t_d, n, 0) \cdot B' = (0, \dots, 0, n, 0)$.

4.1 Experimental Results

To solve the SVP problem we used the BKZ algorithm [35] as implemented in fplll [12]. However, this implementation is only efficient for small block size (say less than 35), due to the fact that BKZ is an exponential algorithm in the block size. Thus for larger block size we implemented a variant of the BKZ-2.0 algorithm [15], however this algorithm is only effective for block sizes β greater than 50. In tuning BKZ-2.0 we used the following strategy, at the end of every round we determined whether we had already solved for the private key, if not we continued, and then gave up after ten rounds. As stated above we applied our attack to the curve **secp256k1**.

We wished to determine what the optimal strategy was in terms of the minimum value of Z we should take, the optimal lattice dimension, and the optimal lattice algorithm. Thus we performed a number of experiments which are reported on in Tables

2 and 3 in Appendix A; where we present our best results obtained for each (d, Z) pair. We also present graphs to show how the different values of β affected the success rate. For each lattice dimension, we measured the optimal parameters as the ones which minimized the value of lattice execution time divided by probability of success. The probability of success was measured by running the attack a number of times, and seeing in how many executions we managed to recover the underlying secret key. We used Time divided by Probability is a crude measure of success, but we note this hides other issues such as expected number of executions of the signature algorithm needed.

All executions were performed on an Intel Xeon CPU running at 2.40 GHz, on a machine with 4GB of RAM. The programs were run in a single thread, and so no advantages where made of the multiple cores on the processor. We ran experiments for the SVP attack using BKZ with block size ranging from 5 to 40 and with BKZ-2.0 with blocksize 50. With our crude measure of Time divided by Probability we find that BKZ with block size 15 or 20 is almost always the method of choice for the SVP method.

We see that the number of signatures needed is consistent with what theory would predict in the case of $Z = 1$ and $Z = 2$, i.e. the lattice reduction algorithm can extract from the side-channel the underlying secret key as soon as the expected number of leaked bits slightly exceeds the number of bits in the secret key. For $Z = 0$ this no longer holds, we conjecture that this is because the lattice algorithms are unable to reduce the basis well enough, in a short enough amount of time, to extract the small amount of information which is revealed by each signature. In other words the input basis for $Z = 0$ is too close to looking like a random basis, unless a large amount of signatures is used.

To solve the CVP problem variant we applied a pre-processing of either fplll or BKZ-2.0. When applying pre-processing of BKZ-2.0 we limited to only one round of execution. We then applied an enumeration technique, akin to the enumeration used in the enumeration sub-routine of BKZ, but centered around the target close vector as opposed to the origin. When a close vector was found this was checked to see whether it revealed the secret key, and if not the enumeration was continued. We restricted the number of nodes in the enumeration tree to 2^{29}, so as to ensure the enumeration did not go on for an excessive amount of time in the cases where the solution vector is hard to find (this mainly affected the experiments in dimension greater than 150). See Tables 4 and 5, in Appendix A, for details of these experiments; again we present the best results for each (d, Z) pair. The enumeration time is highly dependent on whether the close lattice vector is really close to the lattice, thus we see that when the expected number of bits revealed per signature times the number of signatures utilized in the lattice, gets close to the bit size of elliptic curve (256) the enumeration time drops. Again we see that extensive pre-processing of the basis with more complex lattice reduction techniques provides no real benefit.

The results of the SVP and CVP experiments (Appendix A) show that for fixed Z, increasing the dimension generally decreases the overall expected running time. In some sense, as the dimension increases more information is being added to the lattice and this makes the desired solution vector stand out more. The higher block sizes perform better in the lower dimensions, as the stronger reduction allows them to isolate the solution vector better. The lower block sizes perform better in the higher dimensions, as

the high-dimensional lattices already contain much information and strong reduction is not required.

The one exception to this rule is the case of $Z = 2$ in the CVP experiments. In dimensions below 80 the CVP can be solved relatively quickly here, whereas in dimensions 80 up to 100 it takes more time. This can be explained as follows: in the low dimension the CVP-tree is not very big, but contains many solutions. This means that enumeration of the CVP-tree is very quick, but the solution vector is not unique. Thus, the probability of success is equal to the probability of finding the right vector. From dimension 80 upwards, we expect the solution vector to be unique, but the CVP-trees become much bigger on average. If we do not stop the enumeration after a fixed number of nodes, it will find the solution with high probability, but the enumeration takes much longer. Here, the probability of success is the probability of finding a solution at all.

We first note, for both our lattice variants, that there is a wide variation in the probability of success, if we ran a larger batch of tests we would presume this would stabilize. However, even with this caveat we notice a number of remarkable facts. Firstly, recall we are trying to break a 256 bit elliptic curve private key. The conventional wisdom has been that using a window style exponentiation method and a side-channel which only records a distinction between addition and doubling (i.e. does not identify which additions), one would need much more than 256 executions to recover the secret key. However, we see that we have a good chance of recovering the key with less than this. For example, Nguyen and Shparlinksi [30] estimated needing $23 \times 2^7 = 2944$ signatures to recover a 160 bit key, when seven consecutive zero bits of the ephemeral private key were detected. Namely they would use a lattice of dimension 23, but require 2944 signatures to enable to obtain 23 signatures for which they could determine the ones with seven consecutive digits of the ephemeral private key. Note that $23 \cdot 7 = 161 > 160$. Liu and Nguyen [24] extended this attack by using improved lattice algorithms, decreasing the number of signatures required. We are able to have a reasonable chance of success with as little as 200 signatures obtained against a 256 bit key.

In our modification of the lattice attack we not only utilize zero least significant bits, but also notice that the end of a run of zeros tells us that the next bit is one. In addition we utilize all of the run of zeros (say for example eight) and not just some fixed predetermined number (such as four). This explains our improved lattice analysis, and shows that one can recover the secret with relatively high probability with just a small number of measurements.

As a second note we see that strong lattice reduction, i.e. high block sizes in the BKZ algorithm, or even applying BKZ-2.0, does not seem to gain us very much. Indeed acquiring a few extra samples allows us to drop down to using BKZ with blocksize twenty in almost all cases. Note that in many of our experiments a smaller value of β resulted in a much lower probability of success (often zero), whilst a higher value of β resulted in a significantly increased run time.

Thirdly, we note that if one is unsuccessful on one run, one does not need to derive a whole new set of traces, simply by increasing the number of traces a little bit one can either take a new random sample of the traces one has, or increase the lattice dimension used.

We end by presenting in Table 1 the best variant of the lattice attack, measured in terms of the minimal value of Time divided by Probability of success, for the number of signatures obtained. We see that in a very short amount of time we can recover the secret key from 260 signatures, and with more effort we can even recover it from the FLUSH+RELOAD attack applied to as little at 200 signatures. We see that it is not clear whether the SVP or the CVP approach is the best strategy.

Table 1. Combined Results. The best lattice parameter choice for each number of signatures obtained (in steps of 20)

Expected # Sigs	SVP/ SVP	d	$Z = \min\{z_i\}$	Pre-Processing and/or SVP Algorithm	Time (s)	Prob Success	$100\times$ Time/Prob
200	SVP	100	1	BKZ ($\beta = 30$)	611.13	3.5	17460
220	SVP	110	1	BKZ ($\beta = 25$)	78.67	2.0	3933
240	CVP	60	2	BKZ ($\beta = 25$)	2.68	0.5	536
260	CVP	65	2	BKZ ($\beta = 10$)	2.26	5.5	41
280	CVP	70	2	BKZ ($\beta = 15$)	4.46	29.5	15
300	CVP	75	2	BKZ ($\beta = 20$)	13.54	53.0	26
320	SVP	80	2	BKZ ($\beta = 20$)	6.67	22.5	29
340	SVP	85	2	BKZ ($\beta = 20$)	9.15	37.0	24
360	SVP	90	2	BKZ ($\beta = 15$)	6.24	23.5	26
380	SVP	95	2	BKZ ($\beta = 15$)	6.82	36.0	19
400	SVP	100	2	BKZ ($\beta = 15$)	7.22	33.5	21
420	SVP	105	2	BKZ ($\beta = 15$)	7.74	43.0	18
440	SVP	110	2	BKZ ($\beta = 15$)	8.16	49.0	16
460	SVP	115	2	BKZ ($\beta = 15$)	8.32	52.0	16
480	CVP	120	2	BKZ ($\beta = 10$)	11.55	87.0	13
500	CVP	125	2	BKZ ($\beta = 10$)	10.74	93.5	12
520	CVP	130	2	BKZ ($\beta = 10$)	10.50	96.0	11
540	SVP	135	2	BKZ ($\beta = 10$)	7.44	55.0	13

5 Mitigation

As our attack requires capturing multiple signatures, one way of mitigating it is limiting the number of times a private key is used for signing. Bitcoin, which uses the **secp256k1** curve on which this work focuses, recommends using a new key for each transaction [28]. This recommendation, however, is not always followed [34], exposing users to the attack.

Another option to reduce the effectiveness of the FLUSH+RELOAD part of the attack would be to exploit the inherent properties of this "Koblitz" curve within the OpenSSL implementation; which would also have the positive side result of speeding up the scalar multiplication operation. The use of the *GLV method* [18] for point multiplication would not completely thwart the above attack, but, in theory, reduces its effectiveness. The GLV method is used to speed up the computation of point scalar multiplication when the

elliptic curve has an efficiently computable endomorphism. This partial solution is only applicable to elliptic curves with easily computable automorphisms with sufficiently large automorphism group; such as the curve **secp256k1** which we used in our example.

The curve **secp256k1** is defined over a prime field of characteristic p with $p \equiv 1 \mod 6$. This means that $\mathbb{F}_p$ contains a primitive 6th root of unity ζ and if (x,y) is in the group of points on E, then $(-\zeta x, y)$ is also. In fact, $(-\zeta x, y) = [\lambda](x,y)$ for some $\lambda^6 = 1 \mod n$. Since the computation of $(-\zeta x, y)$ from (x,y) costs only one finite field multiplication (far less than computing $[\lambda](x,y)$) this can be used to speed up scalar multiplication: instead of computing $[k]G$, one computes $[k_0]G + [k_1]([\lambda]G)$ where k_0, k_1 are around the size of $k^{1/2}$. This is known to be one of the fastest methods of performing scalar multiplication [18]. The computation of $[k_0]G + [k_1]([\lambda]G)$ is not done using two scalar multiplications then a point addition, but uses the so called *Straus-Shamir* trick which used joint double and add operations [18, Alg 1] performing the two scalar multiplications and the addition simultaneously.

The GLV method alone would be vulnerable to simple side-channel analysis. It is necessary to re-code the scalars k_0 and k_1 and comb method as developed and assembled in [17] so that the execution is regular to thwart simple power analysis and timing attacks. Using the attack presented above we are able to recover around 2 bits of the secret key for each signature monitored. If the GLV method were used in conjunction with wNAF, the number of bits (on average) leaked per signature would be reduced to $4/3$. It is also possible to extend the GLV method to representations of k in terms of higher degrees of λ, for example writing $k = k_0 + k_1\lambda + \cdots + k_t\lambda^t \mod n$. For $t = 2$ the estimated rate of bit leakage would be $6/7$ bits per signature (though this extension is not possible for the example curve due to the order of the automorphism).

We see that using the GLV method can reduce the number of leaked bits but it is not sufficient to prevent the attack. A positive flip side of this and the attack of [39] is that implementing algorithms which will improve the efficiency of the scalar multiplication seem, at present, to reduce the effectiveness of the attacks.

Scalar blinding techniques [10,25] use arithmetic operations on the scalar to hide the value of the scalar from potential attackers. The method suggested by these works is to compute $[(k + m \cdots n + \bar{m})]G - [\bar{m}]G$ where m and $\bar{m}$ are small (e.g. 32 bits) numbers. The random values used mask the bits of the scalar and prevent the spy from recovering the scalar from the leaked data.

The information leak in our attack originates from using the sliding window in the wNAF algorithm for scalar multiplication. Hence, an immediate fix for the problem is to use a fixed window algorithm for scalar multiplication. A naïve implementation of a fixed window algorithm may still be vulnerable to the PRIME+PROBE attack, e.g. by adapting the technique of [33]. To provide protection against the attack, the implementation must prevent any data flow from sensitive key data to memory access patterns. Methods for achieving this are used in NaCL [8], which ensures that the sequence of memory accesses it performs is not dependent on the private key. A similar solution is available in the implementation of modular exponentiation in OpenSSL, where the implementation attempts to access the same sequence of memory lines irrespective of the private key. However, this approach may leak information [7, 37].

Acknowledgements. The first and fourth authors wish to thank Dr Katrina Falkner for her advice and support and the Defence Science and Technology Organisation (DSTO) Maritime Division, Australia, who partially funded their work. The second and third authors work has been supported in part by ERC Advanced Grant ERC-2010-AdG-267188-CRIPTO, by EPSRC via grant EP/I03126X, and by Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under agreement number FA8750-11-2-0079[1].

References

1. Acıiçmez, O.: Yet another microarchitectural attack: exploiting I-Cache. In: Ning, P., Atluri, V. (eds.) Proceedings of the ACM Workshop on Computer Security Architecture, Fairfax, Virginia, United States, pp. 11–18 (November 2007)
2. Acıiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Mangard, S., Standaert, F.-X. (eds.) Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems, Santa Barbara, California, United States, pp. 110–124 (August 2010)
3. Acıiçmez, O., Gueron, S., Seifert, J.-P.: New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In: Galbraith, S.D. (ed.) Cryptography and Coding 2007. LNCS, vol. 4887, pp. 185–203. Springer, Heidelberg (2007)
4. Acıiçmez, O., Koç, Ç.K., Seifert, J.-P.: On the power of simple branch prediction analysis. In: Proceedings of the Second ACM Symposium on Information, Computer and Communication Security, Singapore, pp. 312–320 (2007)
5. Acıiçmez, O., Schindler, W.: A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 256–273. Springer, Heidelberg (2008)
6. Arcangeli, A., Eidus, I., Wright, C.: Increasing memory density by using KSM. In: Proceedings of the Linux Symposium, Montreal, Quebec, Canada, pp. 19–28 (July 2009)
7. Bernstein, D.J.: Cache-timing attacks on AES (April 2005), `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
8. Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: Hevia, A., Neven, G. (eds.) LatinCrypt 2012. LNCS, vol. 7533, pp. 159–176. Springer, Heidelberg (2012)
9. Bos, J.W., Halderman, J.A., Heninger, N., Moore, J., Naehrig, M., Wustrow, E.: Elliptic curve cryptography in practice. Cryptology ePrint Archive, Report 2013/734 (2013), `http://eprint.iacr.org/`
10. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 667–684. Springer, Heidelberg (2009)
11. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 355–371. Springer, Heidelberg (2011)
12. Cadé, D., Pujol, X., Stehlé, D.: Fplll-4.0.4 (2013), `http://perso.ens-lyon.fr/damien.stehle/fplll/`

[1] The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

13. Canteaut, A., Lauradoux, C., Seznec, A.: Understanding cache attacks. Technical Report 5881, INRIA (April 2006)
14. Chen, C., Wang, T., Kou, Y., Chen, X., Li, X.: Improvement of trace-driven I-Cache timing attack on the RSA algorithm. The Journal of Systems and Software 86(1), 100–107 (2013)
15. Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better lattice security estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Heidelberg (2011)
16. Cipresso, T., Stamp, M.: Software reverse engineering. In: Stavroulakis, P., Stamp, M. (eds.) Handbook of Information and Communication Security, vol. 31, pp. 659–696. Springer (2010)
17. Faz-Hernandez, A., Longa, P., Sanchez, A.H.: Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves. Cryptology ePrint Archive, Report 2013/158 (2013), http://eprint.iacr.org/
18. Gallant, R.P., Lambert, R.J., Vanstone, S.A.: Faster point multiplication on elliptic curves with efficient endomorphisms. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 190–200. Springer, Heidelberg (2001)
19. Gopalakrishnan, K., Thériault, N., Yao, C.Z.: Solving discrete logarithms from partial knowledge of the key. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 224–237. Springer, Heidelberg (2007)
20. Gullasch, D., Bangerter, E., Krenn, S.: Cache games — bringing access-based cache attacks on AES to practice. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, California, United States, pp. 490–595 (May 2011)
21. Howgrave-Graham, N., Smart, N.P.: Lattice attacks on digital signature schemes. Designs, Codes and Cryptography 23(3), 283–290 (2001)
22. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
23. Lenstra, A.K., Lenstra Jr., H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), 515–534 (1982)
24. Liu, M., Nguyen, P.Q.: Solving BDD by enumeration: An update. In: Dawson, E. (ed.) CT-RSA 2013. LNCS, vol. 7779, pp. 293–309. Springer, Heidelberg (2013)
25. Möller, B.: Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks. In: Chan, A.H., Gligor, V.D. (eds.) ISC 2002. LNCS, vol. 2433, pp. 402–413. Springer, Heidelberg (2002)
26. Muir, J.A., Stinson, D.R.: On the low Hamming weight discrete logarithm problem for non-adjacent representations. Appl. Algebra Eng. Commun. Comput. 16(6), 461–472 (2006)
27. Naccache, D., Nguyên, P.Q., Tunstall, M., Whelan, C.: Experimenting with faults, lattices and the DSA. In: Vaudenay, S. (ed.) PKC 2005. LNCS, vol. 3386, pp. 16–28. Springer, Heidelberg (2005)
28. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf
29. Nguyen, P.Q., Shparlinski, I.: The insecurity of the digital signature algorithm with partially known nonces. J. Cryptology 15(3), 151–176 (2002)
30. 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), 201–217 (2003)
31. OpenSSL, http://www.openssl.org.
32. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive, 2002:169 (2002)
33. Percival, C.: Cache missing for fun and profit (2005), http://www.daemonology.net/papers/htt.pdf
34. Ron, D., Shamir, A.: Quantitative analysis of the full Bitcoin transaction graph. Cryptology ePrint Archive, Report 2012/584 (2012), http://eprint.iacr.org/

35. Schnorr, C.-P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In: Budach, L. (ed.) FCT 1991. LNCS, vol. 529, pp. 68–85. Springer, Heidelberg (1991)
36. Stinson, D.R.: Some baby-step giant-step algorithms for the low Hamming weight discrete logarithm problem. Math. Comput. 71(237), 379–391 (2002)
37. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks in AES, and countermeasures. Journal of Cryptology 23(2), 37–71 (2010)
38. Waldspurger, C.A.: Memory resource management in VMware ESX Server. In: Culler, D.E., Druschel, P. (eds.) Proceedings of the Fifth Symposium on Operating Systems Design and Implementation, Boston, Massachusetts, United States, pp. 181–194 (December 2002)
39. Yarom, Y., Benger, N.: Recovering OpenSSL ECDSA nonces using the Flush+Reload cache side-channel attack. Cryptology ePrint Archive, Report 2014/140 (2014), `http://eprint.iacr.org/`
40. Yarom, Y., Falkner, K.: FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In: Proceedings of the 23rd USENIX Security Symposium (to appear, 2014)
41. Zhang, Y., Jules, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: Yu, T., Danezis, G., Gligor, V.D. (eds.) Proceedings of the 19th ACM Conference on Computer and Communication Security, Raleigh, North Carolina, United States, pp. 305–316 (October 2012)

A Experimental Results

Table 2. SVP Analysis Experimental Results : $Z = \min z_i = 1$

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	100× Time/Prob
100	BKZ ($\beta = 30$)	200	611.13	3.5	17460
105	BKZ ($\beta = 30$)	210	702.67	7.5	9368
110	BKZ ($\beta = 25$)	220	78.67	2.0	3933
115	BKZ ($\beta = 25$)	230	71.18	3.5	2033
120	BKZ ($\beta = 20$)	240	14.78	1.0	1478
125	BKZ ($\beta = 10$)	250	6.81	1.0	681
130	BKZ ($\beta = 20$)	260	15.12	4.0	378
135	BKZ ($\beta = 25$)	270	57.83	20.0	289
140	BKZ ($\beta = 20$)	280	16.47	9.0	182
145	BKZ ($\beta = 25$)	290	57.63	29.5	195
150	BKZ ($\beta = 20$)	300	19.05	17.0	112
155	BKZ ($\beta = 15$)	310	13.14	13.5	97
160	BKZ ($\beta = 15$)	320	14.00	16.0	87
165	BKZ ($\beta = 15$)	330	15.75	17.5	90
170	BKZ ($\beta = 15$)	340	17.09	23.0	74
175	BKZ ($\beta = 15$)	350	18.14	23.0	78

Table 3. SVP Analysis Experimental Results : $Z = \min z_i = 2$

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	100× Time/Prob
65	BKZ ($\beta = 25$)	260	5.17	2.5	206
70	BKZ ($\beta = 25$)	280	7.93	13.5	58
75	BKZ ($\beta = 25$)	300	13.58	23.5	57
80	BKZ ($\beta = 20$)	320	6.67	22.5	29
85	BKZ ($\beta = 20$)	340	9.15	37.0	24
90	BKZ ($\beta = 15$)	360	6.24	23.5	26
95	BKZ ($\beta = 15$)	380	6.82	36.0	19
100	BKZ ($\beta = 15$)	400	7.22	33.5	21
105	BKZ ($\beta = 15$)	420	7.74	43.0	18
110	BKZ ($\beta = 15$)	440	8.16	49.0	16
115	BKZ ($\beta = 15$)	460	8.32	52.0	16
120	BKZ ($\beta = 10$)	480	6.49	44.0	14
125	BKZ ($\beta = 10$)	500	6.83	45.0	14
130	BKZ ($\beta = 10$)	520	7.06	48.0	14
135	BKZ ($\beta = 10$)	540	7.44	55.0	13

Table 4. CVP Analysis Experimental Results : $Z = \min z_i = 1$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	100× Time/Prob
150	BKZ ($\beta = 15$)	300	32.43	3.0	1081
155	BKZ ($\beta = 15$)	310	33.90	8.0	424
160	BKZ ($\beta = 20$)	320	48.26	13.5	357
165	BKZ ($\beta = 20$)	330	50.97	20.0	255
170	BKZ ($\beta = 15$)	340	39.58	22.0	180
175	BKZ ($\beta = 15$)	350	41.20	26.0	158
180	BKZ ($\beta = 15$)	360	43.50	31.5	138
185	BKZ ($\beta = 15$)	370	44.30	39.5	112
190	BKZ ($\beta = 15$)	380	45.98	42.0	109
195	BKZ ($\beta = 15$)	390	46.15	46.0	100
200	BKZ ($\beta = 15$)	400	45.41	60.5	75
205	BKZ ($\beta = 15$)	410	48.45	65.5	74
210	BKZ ($\beta = 10$)	420	41.89	59.5	70
215	BKZ ($\beta = 15$)	430	49.56	76.0	65
220	BKZ ($\beta = 15$)	440	49.88	86.0	58
225	BKZ ($\beta = 10$)	450	44.58	77.0	58
230	BKZ ($\beta = 15$)	460	53.23	92.0	58
235	BKZ ($\beta = 10$)	470	52.86	88.0	60
240	BKZ ($\beta = 10$)	480	48.37	90.5	53
245	BKZ ($\beta = 10$)	490	49.74	89.5	56

Table 5. CVP Analysis Experimental Results : $Z = \min z_i = 2$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	100× Time/Prob
60	BKZ ($\beta = 25$)	240	2.68	0.5	536
65	BKZ ($\beta = 10$)	260	2.26	5.5	41
70	BKZ ($\beta = 15$)	280	4.46	29.5	15
75	BKZ ($\beta = 20$)	300	13.54	53.0	26
80	BKZ ($\beta = 20$)	320	21.83	17.0	128
85	BKZ ($\beta = 15$)	340	20.08	25.5	130
90	BKZ ($\beta = 20$)	360	23.36	35.0	67
95	BKZ ($\beta = 20$)	380	22.40	52.5	43
100	BKZ ($\beta = 20$)	400	22.95	67.0	34
105	BKZ ($\beta = 20$)	420	21.76	77.0	28
110	BKZ ($\beta = 15$)	440	14.74	81.0	18
115	BKZ ($\beta = 15$)	460	14.82	86.5	17
120	BKZ ($\beta = 10$)	480	11.55	87.0	13
125	BKZ ($\beta = 10$)	500	10.74	93.5	12
130	BKZ ($\beta = 10$)	520	10.50	96.0	11