"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 techqniques to similar side channel information.

1 Introduction

One important task of cryptographic research is to analyse 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 [20]. 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 [27]. In this work we examine the use of spy-processes on the OpenSSL implementation of the ECDSA algorithm.

OpenSSL [26] 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 [36], then adapted to the case of OpenSSL's implementation of ECDSA over binary fields by Yarom and Benger [35]; running on X86 processor architecture. We exploit a property of the X86 processor architecture using the FLUSH+RELOAD cache side-channel attack [35, 36] to partially recover the ephemeral key used in ECDSA.

In Yarom and Benger [35] 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 [19, 25] 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: Cache side-channel attacks have been used against a number of implementations of cryptosystems [1–3, 6, 9, 10, 32, 37]. These attacks often use the PRIME+PROBE technique [32] to target the L1 cache level. Consequently, a spy program and the victim must execute on the same execution core of the processor.

The FLUSH+RELOAD attack, used in this paper, has been used by Yarom and Falkner [36] and Yarom and Benger [35]. The FLUSH+RELOAD attack, in contrast to the PRIME+PROBE attack, targets the last-level-cache, and can, therefore, be mounted between different cores.

The attack used by Gullasch et al. [18] 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 [36], 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 [35], around 95% of the ephemeral private key was recovered when the Montgomery ladder was used for the scalar multiplication step. Until then, Montgomery ladder was not considered to be vulnerable to side-channel attacks. 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 [17, 22, 31] 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 [35] on the Montgomery ladder. However, in our situation a different approach needs to be taken.

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 [19, 25]. 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 [21] or BKZ [30] to solve the lattice problem. Using this methodology Nguyen and Shparlinski [25], 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 [25] 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 utilise the information in the acquired 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.

Since prior work dealt with applying the FLUSH+RELOAD attack to the Montgomery ladder point multiplicationvalgorithm, and this works applies the FLUSH+RELOAD attack to the wNAF point multiplication algorithm,vwe can now cover all of the point multiplication algorithms used in OpenSSL for ECC operations.

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 [23]. The implementation of the **secp256k1** curve in OpenSSL is interesting as it uses the wNAF method for exponentiation, as opposed to the GLV method [15], 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 usesd 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 [5] 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 ECDSA algorithm, and 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.

ECDSA: The ElGamal Signature Scheme [16] is the basis of the US 1994 NIST standard, Digital Signature Algorithm (DSA). The ECDSA is the adaptation of one step of this algorithm from the multiplicative group of a finite field to the group of points on an elliptic curve, and is the signature algorithm using elliptic curve cryptography with widescale deployment. In this section we outline the algorithm, so as to fix notation for what follows:

Parameters: The scheme uses as 'domain parameters', which are parameters which can be shared by a large number of users, an elliptic curve E defined over a finite field $\mathbb{F}_q$ and a point $G \in E$ of a large prime order n. The point G is considered as a generator of the group of points of order n. The parameters are chosen as such are generally believed to offer a (symmetric) security level of $\sqrt{n}$ given current knowledge and technologies. The field size q is usually taken to be a large odd prime or a power of 2. The implementation of OpenSSL uses both cases, but in this paper we will focus on the case of q being a large prime.

Public-Private Key pairs: The private key is an integer α, $1 < \alpha < n-1$ and the public key is the point $Q = [\alpha]G$. Calculating the private key from the public key requires solving the ECDLP, which is believed to be hard in practice for correctly chosen parameters. The most efficient currently known algorithms for solving the ECDLP have a square root run time in the size of the group [14, 34], hence the aforementioned security level.

Signing: Suppose Bob, with private-public key pair $\{\alpha, Q\}$, wishes to send a signed message m to Alice. For ECDSA he follows the following steps:

1. Using an approved hash algorithm, compute $e = Hash(m)$, take h to be the integer (modulo n) given by the leftmost ℓ bits of e (where $\ell = \min(\log_2(n)$, the bitlength of the hash)).
2. Randomly select $k \in \mathbb{Z}_n$.
3. Compute the point $(x, y) = [k]G \in E$.
4. Take $r = x \bmod n$; if $r = 0$ then return to step 2.
5. Compute $s = k^{-1}(h + r \cdot \alpha) \bmod n$; if $s = 0$ then return to step 2.
6. Bob sends (m, r, s) to Alice.

Verification: To verify the signature on the message sent by Bob, Alice performs the following steps.

1. Check that all received parameters are correct, that $r, s \in \mathbb{Z}_n$ and that Bob's public key is valid, that is $Q \neq \mathcal{O}$ and $Q \in E$ is of order n.
2. Using the same hash function and method as above, compute $h = Hash(m) \pmod n$.
3. Compute $\bar{s} = s^{-1} \bmod n$.
4. Compute the point $(x, y) = [h \cdot \bar{s}]G + [r \cdot \bar{s}]Q$.
5. Verify that $r = x \bmod n$ otherwise reject the signature.

ECDSA is a very brittle algorithm in that an incorrectly implemented version of Step 2 of the signing algorithm can lead to catestrophic security weaknesses. For example, an inappropriate reuse of the random integer led to the highly publicised breaking of the Sony PS3 implementation of ECDSA. Knowledge of the random value k, often referred to as the *ephemeral key*, leads to knowledge of the secret key, since given a message/signature pair and the corresponding ephemeral key one can recover the secrey key via the equation

$$\alpha = (s \cdot k - h) \cdot r^{-1}.$$

It is this equation which we shall exploit in our attack, but we shall do this via obtaining side channel information via a spy process. The spy process targets the computationally expensive part of the signing algorithm, namely Step 3.

Scalar multiplication using wNAF: In OpenSSL Step 3 of 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 substraction 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 $2^w - 1$ extra points are stored, with a precomutation cost off $2^{w-2} - 1$ 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.

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 $d \mod 2 = 1$ **then**
    $d_\ell \leftarrow \min(|[d \mod 2^w] - 2^w|, |d \mod 2^w|)$
    $d = d - d_\ell$
  else
    $d_\ell = 0$
  end
  $d = d/2$
  $\ell += 1$
end

Algorithm 1: Conversion to Non-Adjacent Form

Once the integer d has been recoded into wNAF form, the point multiplication can be carried out by Algorothm 2. The occurence 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. Consequently there are less additions required and so the computation cost of the scalar multiplication is less than using the standard double and add method.

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

Algorithm 2: Computation of kG using OpenSSL wNAF

Before ending this section we note some aspects of the algorithm, and how this is exploited in the our attack. A spy process, by monitoring the cache hits/misses, can determine when the branching 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, and so can determine at least one bit of d, and with probability $1/2$ we determine two bits, with probability $1/4$ 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 line 3 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 [7]. For the **secp256k1** curve, the probability of $\lambda=2$ is less than 2^{-125}. Thus 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 [35]. This section discusses the wNAF implementation 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 [35], 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 weakness in the X86 processor architecture, which allows processes to manipulate the cache of other processes [35, 36]. Using the attack, a spy program can trace or monitor memory read and execute access of a victim program to shared memory pages. The spy program only requires read access to the shared memory pages, hence pages containing binary code in executable files and in shared libraries are susceptible to the attack.

By monitoring the victim accesses to specific locations in shared 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 about the data processed by the victim. In particular, the information obtained through monitoring the phases of the square-and-multiply exponentiation (RSA) or the branching of a conditional statement (ECDLP) [35, 36] reveals the secret key data, which is intended to remain secret.

The FLUSH+RELOAD 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, using the `clflush` instruction on the shared code. 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. Two strategies [36] for reducing the probability of a miss are first, to target memory lines that are accessed multiple times during an operation, as done in the attack on RSA; or second, to increase the time slot length, increasing the portion of the time that the spy spends waiting, the approach used to attack ECDLP.

While the FLUSH+RELOAD attack is generic, the details of its application depends on the attacked program. We now proceed to describe the application of FLUSH+RELOAD to the OpenSSL implementation of wNAF scalar multiplication.

To test the attack 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 [12] the OpenSSL library. For the experiment we used the curve **secp256k1** which is used by Bitcoin.

The pre-computed values used in the OpenSSL implementation of wNAF are often generated on the fly. Consequently, they are not in memory shared with other processes. Hence, FLUSH+RELOAD cannot probe for access into the precomputed values. As such, FLUSH+RELOAD can only recover the sequence of doubles and adds, but cannot identify which of the values is added. As discussed above, this sequence reveals an average of two bits per window-size.

To be able to recover the sequence of point additions and doublings, the spy program should distinguish between consecutive doubling operations and must be able to order them with respect to point additions. Our spy program achieves this by setting the time slot to less than half the length of the group operations. With the selected curve, group add operations take 7,928 cycles on average, while group double operation take 7,601 cycles. Setting the time slot to 3,000 cycles ensures that there is an empty time slot within any group operation, allowing our spy to correctly distinguish consecutive doubles.

Recovering the sequence requires probing the add and double routines. However, four limitations of the FLUSH+RELOAD attack may introduce errors to the recovery, which we now address in turn.

- *Missing a time slot due to system activity:* The attacker, generally, does not have control over system activity, and so little can be done to avoid missing time slots. In such a situation we are unable to determine whether an addition or a doubling has occurred. In our experiments this occurred for roughly 0.55%-0.65% of the time slots. This is a low loss rate which has little consequences on the rest of the attack.
- *False positives due to speculative execution:* Speculative execution [33] is a processor optimisation which greedily executes both arms of a conditional branch before the the processor evaluates the condition. When the condition is evaluated, the processor commits to the taken arm, disposing of the pre-computation done for the other. The effect of speculative execution on our attack is that the processor may start executing both the double and the add functions before evaluating the condition based on the value of the wNAF element. This can result in a positive probe on both the add and the double functions, introducing potential errors to the recovery of the operations sequence.

  To protect against the effects of speculative execution, our attack probes memory lines closer to the end of the respective functions. As speculative execution pre-computes only a small number of instructions before evaluating the condition, only those memory lines at the beginning of the functions are accessed before the condition is evaluated and the wrong branch is aborted.
- *Missing a victim access due to an overlap between the victim access and the spy probe:* To reduce the probability of this our attack probes memory lines that are accessed more than once during the computation. The OpenSSL implementation of the add and double functions is, essentially, a sequence of calls to field operations. Memory lines that include a call to a field function are accessed both before the call and when the call returns. Hence, probing a memory line that includes a call to a field function has a high probability of capturing at least one of the accesses to the memory line.
- *Missing access due to the limited resolution of the* FLUSH+RELOAD *attack:* When the two accesses to the memory line are two close together, they may both overlap with the probe. To reduce the probability of this happening, our spy program probes a memory line that calls a field multiply operation. Of the field operations used in the add and double functions, multiply is the longest, taking about 550 cycles in

the curve we have tested. With probes taking 450 cycles, the probability of both accesses overlapping the probe is zero.

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 doublings 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 it's 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 [24, 25], 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 [24, 25] 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 [24, 25] 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 $2^{d+\sum l_i} \cdot n^d$. 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 $2^{d+\sum l_i} \cdot n^(d+1)$, by taking the lattice vector generated by $\mathbf{x}' = (\mathbf{u}, \alpha, -1)$ 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 [30] as implemented in fplll [8]. 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 [11], however this algorithm is only effective for block sizes 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, 3 and 4 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. Of course the measure of Time divided by Probability is a crude measure as this also hides the effect on the 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 are 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 centred 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 5, 6 and 7, 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 $Z = 2$ and 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 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 in Nguyen and Shparlinksi [25] 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 ones with seven consecutive digits of the ephemeral private key. Note, $23 \cdot 7 = 161 > 160$. We are able to have a reasonable chance of success with as little as 200 signatures obtained.

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 pre-determined 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, 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.

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 ths work focuses, recommends using a new key for each transaction [23]. This recommendation, however, is not always followed [29], 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* [15] for point multiplication would not completely thwart the above attack, but, in theory, reduces its effectiveness.

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

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

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 [15]. 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 [15, 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 recode the scalars k_0 and k_1 and comb method as developed and assembled in [13] 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.

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 [35] is that implementing algorithms which will improve the efficiency of the scalar multiplication seem, at present, to reduce the effectiveness of the attacks.

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 [28]. 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 [4]. Another solution is available in the implementation of modular exponentiation in OpenSSL. Both these implementations ensure that the computation performs the same sequence of memory accesses, both to code and to data, irrespective of the value of the secret key.

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, and by EPSRC via grant EP/I03126X.

References

1. Onur Acıiçmez. Yet another microarchitectural attack: exploiting I-Cache. In Peng Ning and Vijay Atluri, editors, *Proceedings of the ACM Workshop on Computer Security Architecture*, pages 11–18, Fairfax, Virginia, United States, November 2007.
2. Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In Stefan Mangard and François-Xavier Standaert, editors, *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems*, pages 110–124, Santa Barbara, California, United States, August 2010.
3. Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In Tal Malkin, editor, *Proceedings of the Cryptographers' Track at the RSA Conference*, pages 256–273, San Francisco, California, United States, April 2008.
4. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 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, pages 159–176, Berlin, Heidelberg, 2012. Springer-Verlag.
5. Joppe W. Bos, J. Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. Cryptology ePrint Archive, Report 2013/734, 2013. `http://eprint.iacr.org/`.
6. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 667–684. Springer-Verlag, 2009.
7. Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security - ESORICS 2011*, volume 6879 of *Lecture Notes in Computer Science*, pages 355–371. Springer-Verlag, 2011.
8. David Cadé, Xavier Pujol, and Damien Stehlé. Fplll-4.0.4. `http://perso.ens-lyon.fr/damien.stehle/fplll/`, 2013.
9. Anne Canteaut, Cédric Lauradoux, and André Seznec. Understanding cache attacks. Technical Report 5881, INRIA, April 2006.
10. CaiSen Chen, Tao Wang, YingZhan Kou, XiaoCen Chen, and Xiong Li. Improvement of trace-driven I-Cache timing attack on the RSA algorithm. *The Journal of Systems and Software*, 86(1):100–107, 2013.
11. Yuanmi Chen and Phong Q. Nguyen. Bkz 2.0: Better lattice security estimates. In *Advances in Cryptology - ASIACRYPT 2011*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011.
12. Teodoro Cipresso and Mark Stamp. Software reverse engineering. In Peter Stavroulakis and Mark Stamp, editors, *Handbook of Information and Communication Security*, chapter 31, pages 659–696. Springer, 2010.
13. Armando Faz-Hernandez, Patrick Longa, and Ana H. Sanchez. 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/`.
14. Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Improving the parallelized pollard lambda search on anomalous binary curves. *Mathematics of Computation*, 69(232):1699–1705, 2000.
15. Robert P. Gallant, Robert J. Lambert, and Scott A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.

16. Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.
17. K. Gopalakrishnan, Nicolas Thériault, and Chui Zhi Yao. Solving discrete logarithms from partial knowledge of the key. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *Progress in Cryptology – INDOCRYPT 2007*, volume 4859 of *Lecture Notes in Computer Science*, pages 224–237. Springer, 2007.
18. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games — bringing access-based cache attacks on AES to practice. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 490–595, Oakland, California, United States, May 2011.
19. Nick Howgrave-Graham and Nigel P. Smart. Lattice attacks on digital signature schemes. *Designs, Codes and Cryptography*, 23(3):283–290, 2001.
20. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
21. A.K. Lenstra, H.W.jun. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515 –534, 1982.
22. James A. Muir and Douglas R. Stinson. On the low hamming weight discrete logarithm problem for nonadjacent representations. *Appl. Algebra Eng. Commun. Comput.*, 16(6):461–472, 2006.
23. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`.
24. Phong Q. Nguyen and Igor Shparlinski. The insecurity of the digital signature algorithm with partially known nonces. *J. Cryptology*, 15(3):151–176, 2002.
25. Phong Q. Nguyen and Igor E. Shparlinski. The insecurity of the elliptic curve digital signature algorithm with partially known nonces. *Designs, Codes and Cryptography*, 30(2):201–217, September 2003.
26. OpenSSL. `http://www.openssl.org`.
27. Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002:169, 2002.
28. Colin Percival. Cache missing for fun and profit. `http://www.daemonology.net/papers/htt.pdf`, 2005.
29. Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph quantitative analysis of the full Bitcoin transaction graph. Cryptology ePrint Archive, Report 2012/584, 2012. `http://eprint.iacr.org/`.
30. Claus-Peter Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In *Fundamentals of Computation Theory – FCT 1991*, volume 529 of *Lecture Notes in Computer Science*, pages 68–85. Springer, 1991.
31. Douglas R. Stinson. Some baby-step giant-step algorithms for the low hamming weight discrete logarithm problem. *Math. Comput.*, 71(237):379–391, 2002.
32. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks in AES, and countermeasures. *Journal of Cryptology*, 23(2):37–71, January 2010.
33. Augustus K. Uht and Vijay Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th International Symposium on Microarchitecture*, pages 313–325, Ann Arbor, Michigan, United States, November 1995.
34. Michael J. Wiener and Robert J. Zuccherato. Faster attacks on elliptic curve cryptosystems. In Stafford E. Tavares and Henk Meijer, editors, *Selected Areas in Cryptography*, volume 1556 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 1998.
35. Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. `http://eprint.iacr.org/`.
36. Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. Cryptology ePrint Archive, Report 2013/448, 2013. `http://eprint.iacr.org/`.
37. Yinqian Zhang, Ari Jules, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *Proceedings of the 19th ACM Conference on Computer and Communication Security*, pages 305–316, Raleigh, North Carolina, United States, October 2012.

A Experimental Results

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	$100\times$ Time/Prob
240	BKZ ($\beta = 25$)	240	212.01	8.0	2125
245	BKZ ($\beta = 20$)	245	50.78	2.5	2031
250	BKZ ($\beta = 20$)	250	52.08	2.5	2083
255	BKZ ($\beta = 20$)	255	53.60	3.0	1786
260	BKZ ($\beta = 20$)	260	52.93	6.0	882
265	BKZ ($\beta = 20$)	265	54.97	8.5	646
270	BKZ ($\beta = 15$)	270	35.48	3.5	1013
275	BKZ ($\beta = 20$)	275	55.30	12.5	442
280	BKZ ($\beta = 20$)	280	58.55	11.5	508
285	BKZ ($\beta = 20$)	285	61.56	16.0	384
290	BKZ ($\beta = 20$)	290	67.47	18.5	364
295	BKZ ($\beta = 15$)	295	43.92	9.5	462
300	BKZ ($\beta = 20$)	300	73.30	20.0	366
305	BKZ ($\beta = 20$)	305	78.09	27.0	289
310	BKZ ($\beta = 20$)	310	83.01	29.0	286
315	BKZ ($\beta = 20$)	315	87.70	29.0	302
320	BKZ ($\beta = 20$)	320	93.28	30.0	310
325	BKZ ($\beta = 20$)	325	91.54	22.0	416
330	BKZ ($\beta = 15$)	330	63.34	21.0	301
335	BKZ ($\beta = 15$)	335	64.28	23.0	279

Table 2. SVP Analysis Experimental Results : $Z = \min z_i = 0$

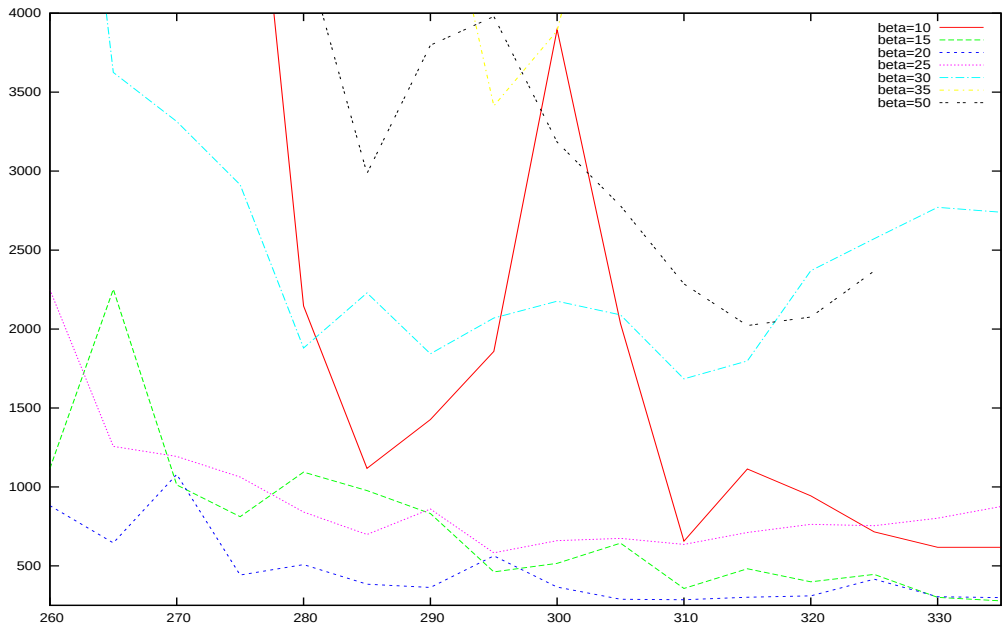

Fig. 1. SVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 0$

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	$100\times$ 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 : $Z = \min z_i = 1$

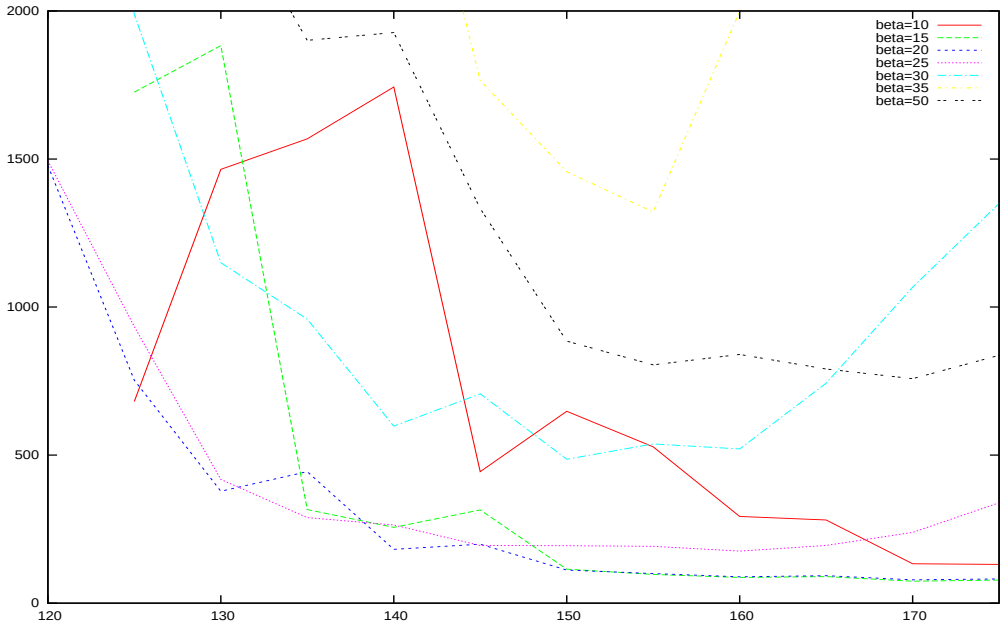

Fig. 2. SVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 1$

d	Algorithm	Expected # Sigs	Lattice Time (s)	Prob.0 Success	$100\times$ 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. SVP Analysis Experimental Results : $Z = \min z_i = 2$

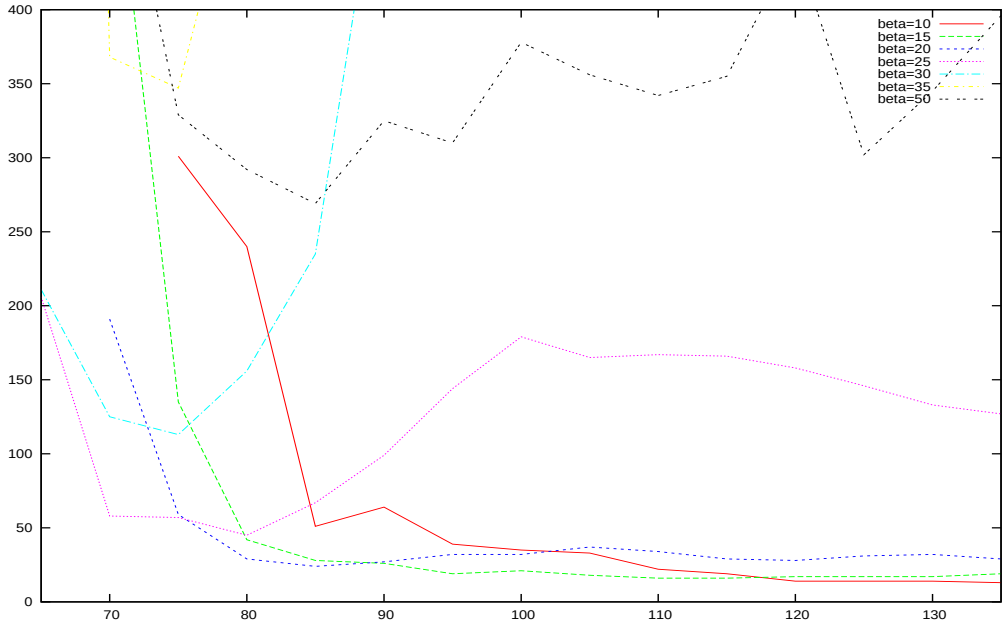

Fig. 3. SVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 2$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	$100\times$ Time/Prob
300	BKZ ($\beta = 10$)	300	100.09	5.0	2002
305	BKZ ($\beta = 20$)	305	186.61	13.5	1382
310	BKZ ($\beta = 10$)	310	110.12	7.0	1573
315	BKZ ($\beta = 10$)	315	114.22	10.0	1142
320	BKZ ($\beta = 10$)	320	125.69	10.5	1197
325	BKZ ($\beta = 20$)	325	246.89	22.5	1097
330	BKZ ($\beta = 15$)	330	153.59	16.0	960
335	BKZ ($\beta = 15$)	335	162.22	24.5	662
340	BKZ ($\beta = 15$)	340	167.08	19.0	879
345	BKZ ($\beta = 15$)	345	178.54	30.0	595
350	BKZ ($\beta = 15$)	350	191.91	30.5	629
355	BKZ ($\beta = 15$)	355	194.37	32.0	607
360	BKZ ($\beta = 15$)	360	198.39	34.0	583
365	BKZ ($\beta = 15$)	365	216.43	44.5	486
370	BKZ ($\beta = 15$)	370	218.68	44.5	491
375	BKZ ($\beta = 15$)	375	228.25	45.5	502
380	BKZ ($\beta = 10$)	380	187.14	40.0	468
385	BKZ ($\beta = 15$)	385	243.71	58.0	420
390	BKZ ($\beta = 15$)	390	249.26	61.0	409
395	BKZ ($\beta = 10$)	395	213.76	43.5	491

Table 5. CVP Analysis Experimental Results : $Z = \min z_i = 0$

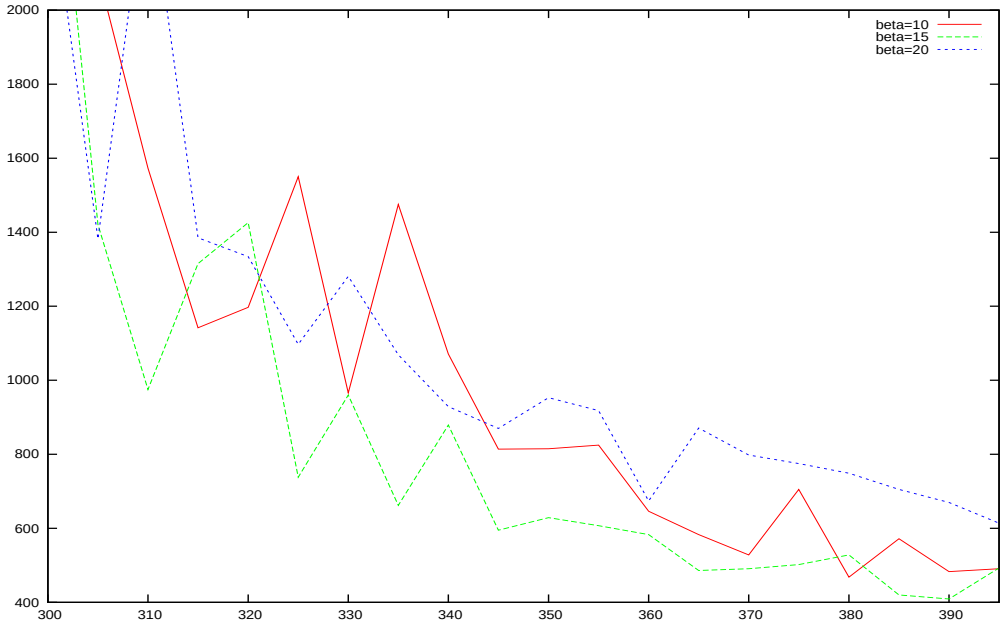

Fig. 4. CVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 0$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	$100\times$ 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 6. CVP Analysis Experimental Results : $Z = \min z_i = 1$

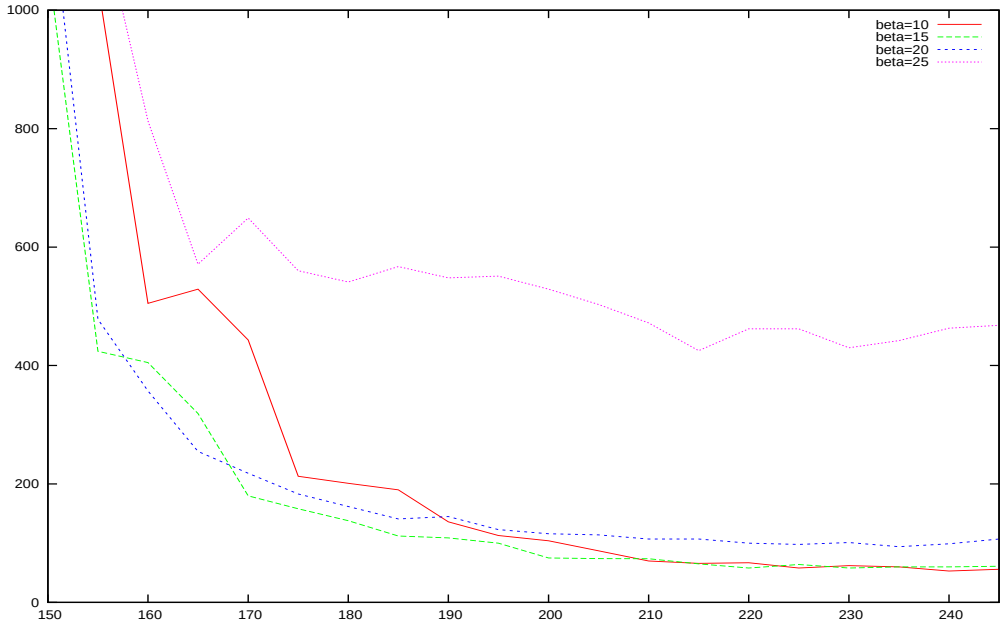

Fig. 5. CVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 1$

d	Pre-Processing Algorithm	Expected # Sigs	Time (s)	Prob.0 Success	$100\times$ 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

Table 7. CVP Analysis Experimental Results : $Z = \min z_i = 2$

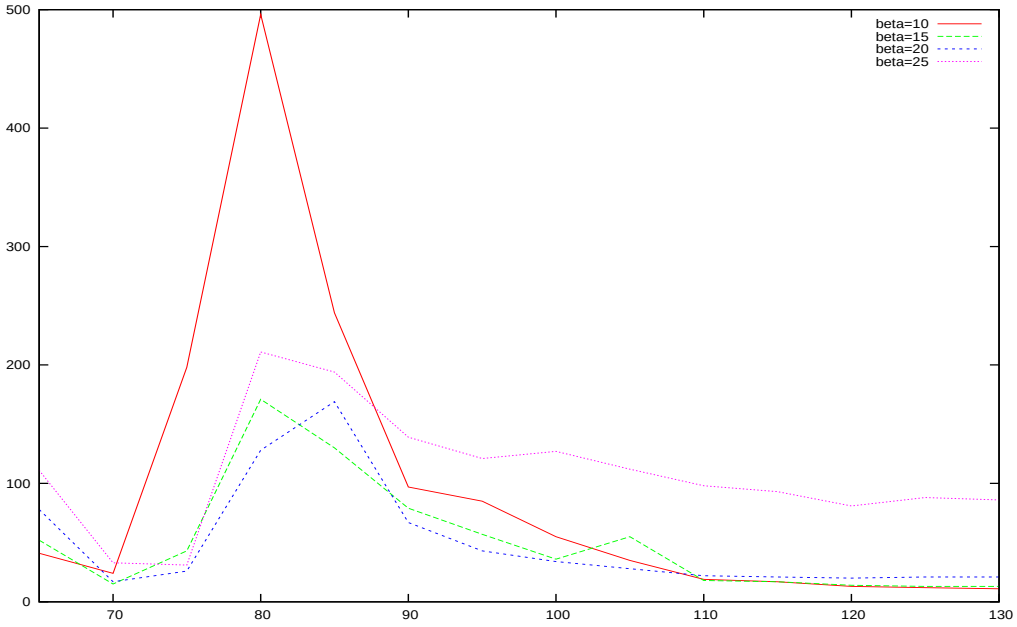

Fig. 6. CVP Experiments: d vs Time/Prob for various β and $Z = \min z_i = 2$