Another Lattice Attack against the wNAF Implementation of ECDSA to Recover More Bits per Signature

No Author Given

No Institute Given

Abstract. This paper presents a practical lattice attack on ECDSA which is implemented with the windowed Non-Adjacent-Form (wNAF) representations to compute the scalar multiplication over elliptic curves. Compared with the existing lattice attacks in the literature, our method recovers the private key from side channels by extracting more information per signature. In particular, in the scalar multiplications, we monitor the invert function through cache side channels, in addition to the functions of double and add which are only exploited in the literature. By monitoring the invert function, we obtain the signs of the wNAF representations of an ephemeral key. The "double-add-invert" chain obtained from the cache side channels, is disposed to extract the information of the ephemeral key at the position of the non-zero digits. Theoretically, 153.2 bits of the ephemeral key will be extracted per signature for 256-bit ECDSA on a prime field, much more than existing methods. Then, we convert the problem of recovering the private key based on the extracted bits, to the Hidden Number Problem (HNP) which is solved by lattice reduction algorithms. We implemented the Flush+Flush cache-based side channel and applied it to ECDSA with the secp256k1 curve in OpenSSL 1.1.1b, to monitor the functions of double, add and invert. In the experiments, the BKZ lattice reduction algorithm is used to solve the Shortest Vector Problem (SVP) of HNP instances. The experimental results show that the private key is successfully recovered from 60 signatures, if the Flush+Flush side channel is built without any error. To the best of our knowledge, this is the first time to obtain and exploit the signs of the wNAF representations in the side channel attacks against ECDSA, and we demonstrate the potential to recover the private key from fewer signatures.

Keywords: ECDSA · windowed Non-Adjacent-Form · OpenSSL · Lattice Attack · Hidden Number Problem · Cache Side Channel

1 Introduction

The Elliptic Curve Digital Signature Algorithm (ECDSA) [8,30] is a digital signature algorithm over elliptic curves, widely used in many popular applications, such as TLS [51], OpenPGP [29], smart card [47], and Bitcoin [35]. The core operation of ECDSA is the scalar multiplication of a base point (or generator) over elliptic curves by a random nonce (or ephemeral key), and its semantical security relies on the computational intractability to find the ephemeral key for any given pair of a scalar multiplication and a base point, i.e., the elliptic curve discrete logarithm problem (ECDLP).

However, there are side channel attacks to obtain the information on the ephemeral key, during the scalar multiplication. As long as some bits of the ephemeral key are leaked, the private key will be recovered [26,38–40]. Howgrave-Graham and Smart [26] theoretically showed that DSA was vulnerable to such partial ephemeral key exposures in 2001. Then, Nguyen and Shparlinski [39] improved their method with further details and proposed a provable polynomial-time attack. More importantly, they extended this theoretical attack from DSA to ECDSA [40]. The basic idea is to reduce the problem of private key recovery to an instance of the Hidden Number Problem (HNP), which is further reduced to the Closest Vector Problem (CVP) or the Shortest Vector Problem (SVP) in a lattice, with the knowledge of consecutive bits of the ephemeral keys. Brumley et al. employed this lattice attack to recover the ECDSA private keys from leaked least significant bits (LSBs) [13] or leaked most significant bits (MSBs) [14].

With the Flush+Reload cache side channels [56], Benger et al. [9] obtained the position of the least non-zero digit to infer the LSBs of the ephemeral key. This side channel exploits the windowed Non-Adjacent Form (wNAF) implementation [21,31,34,50] for the scalar multiplication of a known point G by the random ephemeral key k. This exploitation works for the elliptic curves over a prime field $\mathbb{F}_p$ in ECDSA, which is implemented in OpenSSL [2] and other cryptographic engines. Then, by constructing an HNP instance solved by the lattice reduction algorithm, they recovered the ECDSA private key from about 200 signatures. In 2015, Van de Pol et al. improved this attack by a more effective way of exacting information from the cache side channel [46], which exploited the information about the positions of higher half non-zero digits of the wNAF representations of the ephemeral key. In the same year, Cao et al. [15] presented two lattice-based differential fault attacks against ECDSA with wNAF implementations. In 2016, Goudarzi et al. [22] extended this attack to the blinded ephemeral keys, which were protected by the addition of a random multiple of the elliptic-curve group order or by a random Euclidean splitting. Then, Fan et al. proposed a new way of extracting and utilizing information obtained from the Flush+Reload side channels. The problem of recovering the secret key is then transformed to the Extended Hidden Number Problem (EHNP) which is also solved by the lattice reduction algorithm. The number of signatures needed is reduced to 4 to recover the private key [19]. In 2017, Wang et al. [54] presented another lattice attack on ECDSA using a small fraction of information from the ephemeral key. They exploited the positions of two non-zero digits together with the length of the wNAF representation of the ephemeral key to construct an HNP instance. They need 85 signatures to recover the private key.

The above works extracted the information in the wNAF representation of the ephemeral key. They tried to make better use of the data achieved through the side channels and construct more effective lattice attacks to recover the private key. They extracted the least non-zero digit [9], half non-zero digits [46], all non-zero digits [19] or two non-zero digits with the total length [54]. In summary, all information exploited in these attacks are the positions of the non-zero digits, and no more extra information about the ephemeral key is extracted from the side channels. Although the lattice attacks are different in these works, no effort and improvement have been made to achieve and exploit more information.

In our paper, we also focus on ECDSA with the wNAF representation of the ephemeral key. We propose a new method to recover the ECDSA private key from the information leaked through side channels. We try to not only make better use of the information achieved from side channels, but also to extract different information from the side channels. First, we analyse the implementation code of ECDSA in OpenSSL and find that the invert function is another exploitable vulnerability in the implementation of the scalar multiplication. It inverts a number so that the subtraction is replaced by addition and the space storing the precomputed points is reduced by half. In the calculation, if the sign of current non-zero digit of the ephemeral key is opposite to the previous one, the invert function is called, and the absolute value of this digit is used to index the precomputed points. Otherwise, the invert function is not called. It helps us to determine the sign of the non-zero digits of the wNAF representations. So we obtain the sign of the non-zero digits in the wNAF representations of the ephemeral key by adding a monitor to the invert function. Thus, we achieve more information about the ephemeral key compared to previous attacks. After that, we manage to recover the consecutive bits at the position of every non-zero digits of the ephemeral key based on the information obtained from the cache side channels. Finally, we construct an HNP [11] instance making use of the consecutive bits of the ephemeral key. This problem is solved by lattice reduction algorithms through converting to the SVP.

This attack is applied to the secp256k1 curve in OpenSSL 1.1.1b in this paper. We choose the Flush+Flush [23] attack to monitor the functions of double, add and invert. By monitoring the invert function in addition to the double and add functions in OpenSSL, we successfully extract whether each digit of the ephemeral key is zero or not, and determine the signs of the non-zero digits. Then we use the BKZ [49] algorithm to solve the HNP instance, and effectively recover the ECDSA private key. If the result of the Flush+Flush attack is perfect, we need about 60 signatures to recover the private key with a probability of 3%.

Through the cache side channel, we get the information about the positions and the signs of all non-zero digits of the ephemeral key. Compared to previous works, our method obtains more information about the ephemeral key, i.e. the signs of all non-zero digits. We extract 153.2 bits on average per signature for 256-bit ECDSA. In theory, the number of signatures that required is 2 if a suitable lattice is constructed. To make better use of the sign information of the non-zero digits, we exploit the information obtained about the wNAF representation to extract sequences of consecutive bits of k to construct the lattice attack, because the steps in existing works do not work for this information. We have to obtain multiple consecutive bits of the ephemeral key by converting the wNAF representation to the binary representation. And then, we use these consecutive bits to construct the HNP instance. To the best of our knowledge, this is the first time to obtain and exploit the signs of the wNAF representations in the side channel attacks against ECDSA, and previous works cannot be straightforwardly applied to this information. Our contributions are summarized as follows:

- We present a new lattice attack to recover the private key of ECDSA with the wNAF representation. First, we improve the cache side channel attacks by monitoring the invert function. It is the first work to exploit the sign of the non-zero digits of the ephemeral key in side channel attacks. Second, for the data from the cache side channel, we do not use them to construct the HNP instance directly, as in previous works. We obtain multiple consecutive bits of the ephemeral key by converting the wNAF representation to the binary representation. Finally, we use these consecutive bits to construct the HNP instance. The private key is recovered by solving the HNP instance using lattice reduction algorithm. These steps are different from existing works.
- We apply our method to the secp256k1 curve in the latest version of OpenSSL. Through the cache side channel, 153.2 bits of information per signature are obtained on average. The experiments show that 60 signatures are enough to recover the private key with a probability of 3%.

The rest of this paper is organized as follows. Section 2 presents some preliminaries. Section 3 provides the details about our attack, and Section 4 shows the implementation details and the experimental results. Section 5 contains some extended discussions. Section 6 introduces some related works. Section 7 draws the conclusion.

2 Preliminaries

In this section, we present the background knowledge about the attack. First we describe the ECDSA and its implementation using the wNAF representation in OpenSSL. Then we explain the attack method of the cache side channels. Also, the hidden number problem and its lattice attack are introduced to dispose the data obtained from cache side channels.

2.1 The Elliptic Curve Digital Signature Algorithm

ECDSA [8, 30] is the adaption of one step of the Digital Signature Algorithm (DSA) [36] from the multiplicative group of a finite field to the group of points on an elliptic curve.

Let E be an elliptic curve defined over a finite field $\mathbb{F}_p$ where p is prime. $G \in E$ is a fixed point of a large prime order q, that is G is the generator of the group of points of order q. These curve and point parameters are publicly known. The private key of ECDSA is an integer α that satisfies $0 < \alpha < q$, and the public key is the point $Q = \alpha G$. Given a hash function h, the ECDSA signature of a message m is computed as follows:

Algorithm 1 Conversion to wNAF Representation

```
Input: Scalar $k$, window size $w$
Output: $k$ in wNAF: $k_0$, $k_1$, $k_2$, ...
$i \leftarrow 0$
while $k > 0$ do
    if $k \bmod 2 = 1$ then
        $k_i \leftarrow k \bmod 2^{w+1}$
        if $k_i \geq 2^w$ then
            $k_i \leftarrow k_i - 2^{w+1}$
        end if
        $k \leftarrow k - k_i$
    else
        $k_i \leftarrow 0$
    end if
    $k \leftarrow k/2$
    $i \leftarrow i + 1$
end while
```

1. Select a random ephemeral key $0 < k < q$.
2. Compute the point $(x, y) = kG$, and let $r = x \bmod q$; if $r = 0$, then go back to the first step.
3. Compute $s = k^{-1}(h(m) + r \cdot \alpha) \bmod q$; if $s = 0$, then go back to the first step.

The pair (r, s) is the ECDSA signature of the message m. For ECDSA signature, the ephemeral key k must be kept random and secret. The equation in the third step shows the private key can be computed if k is leaked. Even if a portion of k is known, the private key can be recovered by lattice attacks. Therefore, the target of most attackers is the scalar multiplication kG expecting to get some effective bits information about k.

2.2 The Scalar Multiplication using wNAF Representation

G is a point defined on the elliptic curve over the finite field, and the scalar k is a big integer. Scalar multiplication kG on the elliptic curve means that the point G is added to itself k times. There are several algorithms to implement the scalar multiplication. In OpenSSL, scalar multiplication in the prime field is implemented using the wNAF representation [21,31,34,50] of the scalar k. In wNAF, a number k is represented by a sequence of digits which value is either zero or an odd number satisfied $-2^w < k_i < 2^w$, where w is the window size. In this representation, any continuous non-zero digit interval is with at least w zero digits. The value of k can be expressed as $k = \sum 2^i k_i$. Algorithm 1 introduces the concrete method for converting a scalar into its wNAF representation.

When computing the scalar multiplication kG, first, a window size w is chosen. Then precomputation and storage of the points $\{\pm G, \pm 3G, ..., \pm(2^{w-1})G\}$ are executed. After converting k to the wNAF form, the multiplication kG is executed as described in Algorithm 2.

Algorithm 2 Implementation of kG Using wNAF

```
Input: Scalar $k$ in wNAF: $k_0$, $k_1$, ..., $k_{l-1}$ and precomputed points
    $\{\pm G, \pm 3G, ..., \pm(2^{w-1})G\}$
Output: $kG$
1: $Q \leftarrow G$
2: for $i$ from $l-1$ to 0 do
3:     $Q \leftarrow 2 \cdot Q$
4:     if $k_i \neq 0$ then
5:         $Q \leftarrow Q + k_i G$
6:     end if
7: end for
```

From the algorithm, we can find that the if-then block (Line 4) is vulnerable to side channel attacks. An attacker can use a spy process to monitor the conditional branches through side channels. Then he can get a “double-add” chain, and through it, whether the value of k_i is zero can be inferred. The attacker can use this information to recover the private key.

In OpenSSL, the bit length of k is set to a fixed value$\lfloor \log_2 q \rfloor + 1$ of by adding itself q or $2q$, which can resist the remote timing side channel attack [14]. In most cases, the multiplication is done as $(k+q)G$, but it is still vulnerable in lattice attacks [22].

Also, OpenSSL uses the modified wNAF representation instead of the generalized one as stated in Algorithm 1 to avoid length expansion in some cases and to make more efficient exponentiation. The representation of modified wNAF is very similar to the wNAF. Each non-zero coefficient is followed by at least w zero coefficients, except for the most significant digit which is allowed to violate this condition in some cases. As the use of modified wNAF affects the attack results little, we only consider the case of the wNAF for simplification.

2.3 The Scalar Multiplication in OpenSSL

In OpenSSL 1.1.1b [2], the scalar multiplication with wNAF representation is implemented in the function `ec_wNAF_mul()`. The core computation of the function is shown in Algorithm 3.

In this function, it iterates the k from the most significant digit to the least significant digit in its wNAF representation. In each digit, it performs a double operation (the `EC_POINT_dbl()` function). If the digit is not zero, it runs into the if-then block in Line 6, and first determines whether an invert function is needed to execute. The invert function is to compute the inverse of a number. If the sign of the non-zero digit k_i is opposite to the previous non-zero digit, the invert operation (`EC_POINT_invert()`) is performed (Line 13), which makes it only need the precomputation and storage of the points $\{G, 3G, ..., (2^{w-1})G\}$ and saves a half of storage space. Then it performs an addition operation (the `EC_POINT_add()` function) with indexing from the precomputed points using the absolute value of the digit.

Algorithm 3 The Implementation of The Scalar Multiplication in OpenSSL

```
Input: Scalar $k$ in wNAF $\{k_0, k_1, ..., k_{l-1}\}$ and precomputed points $\{G, 3G, ..., (2^w - 1)G\}$
Output: $kG$
$r \leftarrow 0$, $is\_neg \leftarrow 0$, $r\_is\_inverted \leftarrow 0$
for $i$ from $l - 1$ to 0 do
    if $r \neq 0$ then
        $EC\_POINT\_dbl(r)$   // double
    end if
    if $k_i \neq 0$ then
        $is\_neg \leftarrow (k_i < 0)$
        if $is\_neg$ then
            $k_i \leftarrow -k_i$
        end if
        if $is\_neg \neq r\_is\_inverted$ then
            if $r \neq 0$ then
                $EC\_POINT\_invert(r)$   // invert
            end if
            $r\_is\_inverted \leftarrow !r\_is\_inverted$
        end if
        if $r = 0$ then
            $r \leftarrow EC\_POINT\_copy(k_iG)$
        else
            $r \leftarrow EC\_POINT\_add(r, k_iG)$   // add
        end if
    end if
end for
return r
```

From Algorithm 3, it can be found that two conditional branches are vulnerable. In each loop, the double operation is always performed. But the add operation is only performed if the digit is not zero and the invert operation is only performed if the sign of the digit is opposite to previous one. Therefore, if a spy process can obtain the execution sequence of the double and addition operations while this function is running, one can determine whether each digit of k is zero or not according to this sequence. Also, the execution sequence of the invert operation can be used to determine the sign of the non-zero digits combining the former sequence.

2.4 Cache Side Channel Attacks

Cache side-channel attacks take advantage of the characteristic of cache activity that accessing data from caches is much faster than from memory. Attackers exploit these time variations to deduce the operations of the target process and then infer the key information. Many attack methods are proposed [10, 12, 24, 42, 56] since it is demonstrated feasible in theory [43]. We introduce two typical access-driven attacks called Flush+Reload [56] and Flush+Flush [23] that can be used to monitor the functions of the implementation of ECDSA.

The Flush+Reload attack [56] employs a spy process to monitor whether the specific memory lines have been accessed or not by the victim process. So this attack relies on shared memory between the spy and the victim processes. For attacks in the same machine, the spy can map the victim program file, the victim data file or shared libraries into its own address space to share these pages with the victim. While in the virtualization environments, the page de-duplication technique of the VMM ensures the page sharing between the spy and the victim.

The execution of such an attack consists of three phases:

- **Flush:** In this phase, the attacker uses the `clflush` instruction to flush the desired memory lines out from the caches. This ensures that these lines are accessed from the memory instead of the caches for the next time.
- **Wait:** In this phase, the attacker waits a moment while the victim runs.
- **Reload:** This phase detects whether the victim accesses the memory lines flushed in the first phase during the waiting time. The attacker accesses the desired memory lines to reload them into caches, and measures the time. From the reload time, the attacker determines whether the memory lines are accessed by the victim. If it is longer, it means that the attacker reloads the data from the memory and the victim does not access the data. Otherwise, it means that the victim accesses the data.

The Flush+Flush attack [23] also assumes shared memory. It relies on the execution time of the `clflush` instruction, which is affected by whether the to-be-flushed data are cached or not. The execution time of `clflush` is shorter if the data are not cached and longer if the data are cached. So according to this time, the attackers determine the victim's cache activities.

In general, the Flush+Flush attack also consists of three phases. The first two phases are the same as in the Flush+Reload attack. But in the third phase, it flushes the cache again and measures the flush time instead of the reload time. As the third phase flushes the caches, it doubles as the first phase for subsequent observations.

The execution time of `clflush` is less than the reload time on average, and the first phase and the third phase of the next round are merged together in the Flush+Flush attack. Therefore the Flush+Flush attack is more efficient than the Flush+Reload attack. Thus, the Flush+Flush attack has a better accuracy.

2.5 The Hidden Number Problem and Lattice Attack

The Hidden Number Problem (HNP) is first presented by Boneh and Venkatesan [11] in 1996. It is used to recover the secret key of Diffie-Hellman key exchange, DSA and ECDSA, given some leaked consecutive bits of the ephemeral key.

Given a prime number q and a positive l, and let $t_1, t_2, ..., t_d$ be randomly chosen, which are uniformly and independently in $\mathbb{F}_q$. The HNP can be stated as follows: recovering an unknown number $\alpha \in \mathbb{F}_q$ such that the known number pairs (t_i, u_i) satisfy $v_i = |\alpha t_i - u_i|_q \leq q/2^{l+1}$ for $1 \leq i \leq d$, where $|\cdot|_q$ denotes the reduction modulo q into range $[-q/2, ..., q/2)$. If $|\alpha t - u|_q \leq q/2^{l+1}$ is satisfied, the integer u represents the l most significant bits of αt which is defined as $MSB_l(\alpha t)$.

The HNP problem can be converted to the CVP/SVP problem in lattices and solved by the lattice reduction. Here we provide a brief introduction. For more details on lattice, please refer to the literature [41]. Consider the Euclidean space $\mathbb{R}^d$ and let $B = \{\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_z}\}$ be a set of linearly independent vectors in $\mathbb{R}^d$. The set of vectors

$$L = L(B) = \{\sum_{i=1}^{z} \beta_i \mathbf{b_i} \mid \beta_i \in \mathbb{Z}\}$$

is the lattice generated by B. The set B is called a basis of L, and L is spanned by B. The number z representing the number of vectors in B is the dimension or rank of $L(B)$. If $z = d$, the lattice $L(B)$ is a full-dimension lattice. As the lattice is a set of vectors, the norm of the shortest non-zero vector is called the first minima and denoted by $\lambda_{\mathbf{1}}(L)$. That is, $\lambda_{\mathbf{1}}(L) = \min\{\|\mathbf{u}\| \mid 0 \neq \mathbf{u} \in L\}$, where $\|\mathbf{u}\|$ denotes the Euclidean norm of the vector $\mathbf{u}$. The problem of finding a non-zero vector $\mathbf{v} \in L$ with the minimal norm is called the shortest vector problem (SVP). While for a lattice L and an arbitrary vector $\mathbf{v} \in \mathbb{R}^d$, the problem of finding a lattice vector $\mathbf{u} \in L$ of minimal distance from $\mathbf{v}$ is called the closest vector problem (CVP) similarly. In other words, finding a vector $\mathbf{u}$ satisfied $\|\mathbf{u}\| = \min\{\|\mathbf{u} - \mathbf{v}\| \mid \mathbf{u} \in L\}$.

Exploiting lattices to solve the HNP problem, we construct a d+1 dimensional lattice $L(B)$ spanned by the rows of the following matrix:

$$B = \begin{pmatrix} q & 0 & \cdots & 0 & 0 \\ 0 & q & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & q & 0 \\ t_1 & \cdots & \cdots & t_d & 1/2^{l+1} \end{pmatrix}.$$

Considering the vector $\mathbf{h} = (\alpha t_1 \bmod q, ..., \alpha t_d \bmod q, \alpha/2^{l+1})$, it can be obtained by multiplying the last row vector of B by α and then subtracting appropriate multiples of the first d row vectors. Thus the vector $\mathbf{h}$ belongs to $L(B)$. We call $\mathbf{h}$ the hidden vector because the last coordinate of $\mathbf{h}$ discloses the hidden number α. Let the vector $\mathbf{u} = (u_1, ..., u_d, 0)$, so the distance between $\mathbf{h}$ and $\mathbf{u}$ is $\|\mathbf{h} - \mathbf{u}\| \leq q\sqrt{d+1}/2^{l+1}$. While the lattice determinant of $L(B)$ is $q^d/2^{l+1}$, thus the vector $\mathbf{h}$ is very close to the vector $\mathbf{u}$. Solving the CVP problem with input B and $\mathbf{u}$ reveals the vector $\mathbf{h}$, hence the private key α is recovered.

Because solving the CVP instance requires exponential time in the lattice rank, we can use the embedding technique [37] to transform it to an SVP instance which just requires the polynomial time to solve. We construct a d+2 dimensional lattice $L(B')$ spanned by the rows of the matrix

$$B' = \begin{pmatrix} B & 0 \\ \mathbf{u} & q/2^{l+1} \end{pmatrix}.$$

Similarly, the vector $\mathbf{h}' = (\alpha t_1 - u_1 \bmod q, ..., \alpha t_d - u_d \bmod q, \alpha/2^{l+1}, -q/2^{l+1})$ belongs to the lattice $L(B')$. Its norm satisfies that $\|\mathbf{h}'\| \leq q\sqrt{d+2}/2^{l+1}$, while the lattice determinant of $L(B')$ is $q^{d+1}/2^{2l+2}$. This indicates that the vector $\mathbf{h}'$ is a very short vector. Note that this lattice also contains another vector $(-t_1, ..., -t_d, q, 0) \cdot B = (0, ..., 0, q/2^{l+1}, 0)$, which is most likely the shortest vector of the lattice. So we expect the second vector in a reduced basis of the lattice is equal to $\mathbf{h}'$ with a "good" chance for a suitably strong lattice reduction algorithm. Then we can acquire the hidden number α by solving the SVP problem.

For the ECDSA algorithm implemented by OpenSSL with the wNAF, the attackers can obtain some consecutive bit fragments of the ephemeral key k through the cache side channels. Then, the problem of recovering the secret key can be transformed to the HNP problem. The attackers can recover the secret key by solving the HNP problem using lattice reduction algorithms.

3 Attacking ECDSA

This section proposes a new method to recover the private key of ECDSA. First, we analyse the invert function in the scalar multiplication with wNAF representation in the ECDSA algorithm. Then we use the invert function for improving the cache side channel attacks, from which the sign of the non-zero digits of k is determined.

After that, we make use of the obtained information to recover the consecutive bits at the position of every non-zero digits of k. Finally, we construct an HNP instance by the consecutive bits and transform it to the problem of solving the SVP/CVP in some lattice with the lattice reduction algorithms. Note that the steps in existing works do not work for the signs of the non-zero digits. So we have to first obtain consecutive bits of the ephemeral key by converting the w-NAF representation to the binary representation. Then, we use these consecutive bits to construct the HNP instance.

3.1 Attacking wNAF through the Cache Side Channels

First, recalling the implementation of scalar multiplication in OpenSSL, it uses the invert function to compute the inverse element of a number. Applying this function, the space of precomputed points is saved by half, only need to store $\{G, 3G, ..., (2^w - 1)G\}$. As shown in Algorithm 3, it iterates k from the most significant digit to the least significant digit. In each digit, it performs a double function. While if the digit is not zero, it first determines whether the invert function is executed. If the sign of the digit is opposite to the prior one, the invert function is called. Then it indexes from the precomputed points using the absolute value of the digit and performs an addition operation.

Originally, the vulnerability comes from the double and add function. The double function is called at every digit, and this reveals the digit sequence. The add function is called just when the digit is non-zero, which makes it possible to distinguish whether the digit is zero or not. That reveals the position of all non-zero digits. However, the invert function is also vulnerable, because it is called conditionally. Only when the sign of the non-zero digit is opposite to the previous, the invert function will be performed. Because the first non-zero digit is positive, the signs of all the non-zero digit are deduced based on the execution of the invert function.

We use a spy process to monitor one memory line of the code of the double, add and invert functions during the scalar multiplication. The time is divided into slots, and in each slot, the spy determines whether the three functions are performed or not by monitoring the cache hits/misses. Then we obtain a "double-add-invert" chain. According to the "double" and "add" in this chain, we determine whether each digit of k is zero or not, as done in previous works. Also, based on the "invert" in this chain, we infer the sign of each non-zero digit.

In the OpenSSL library (especially the latest OpenSSL 1.1.1b), the functions of double, add and invert are implemented as `EC_POINT_dbl()`, `EC_POINT_add()` and `EC_POINT_invert()`. The existing attacks monitor `EC_POINT_dbl()` and `EC_POINT_add()`. In our attack, we improve the original cache side channel attack by adding a new monitor to the `EC_POINT_invert()` function. We use a spy process to monitor one memory line of each function. In each time slot the spy determines whether three functions are performed or not. Then we get a new "double-add-invert" chain through the cache side channel attack.

When we use the "double-add-invert" chain to extract the digits of k, the "double" represents the double function is called, and the "add" represents both double and add are called. The "invert" represents the invert function is called. Therefore, the "double" appears meaning that k_i is zero and the "add" appears meaning that k_i is not zero. We use the "invert" to determine the sign of k_i. The sign of k_i is related to the previous non-zero digit. First, the "invert" comes out together with "add". If the "invert" appears, it represents that the sign of k_i is opposite to the previous non-zero digit. While, if the "invert" does not appear when "add" comes out, it represents that the sign of this digit is the same as the previous non-zero one.

Because the wNAF representation of k, the most significant digit is always positive. Thus we can determine the sign of all non-zero digits. In this way, we obtain all the positions of the non-zero digits and the signs of them.

3.2 Recovering Consecutive Bits

In this section, we introduce how to recover the consecutive bits at the position of every non-zero bits for the ephemeral key k, exploiting the information obtained from the side channel.

First, we denote the wNAF representation of k as $k = \sum k_i 2^i$, and the binary representation as $k = \sum b_i 2^i$. From the Algorithm 1 we know that the position of k_i in the wNAF representation is the same as b_i in the binary representation.

When we know the information about whether k_i is zero and the sign of the non-zero k_i, we can simply determine some bits of k. For example, if we obtain the sign of the least non-zero k_j, we can infer that b_j is one and b_i is zero for $0 \leq i < j$. But for arbitrary non-zero digits, it can not determine whether the bit is zero or one.

Let m and $m+n$ be the positions of two consecutive non-zero digits of the wNAF representation, and w be the window size. That is, $k_m, k_{m+n} \neq 0$ and $k_{m+i} = 0$ for all $0 < i < n$. We analyse the transformation method between the binary and wNAF representation, getting the following result:

$$b_{m+n} = \begin{cases} 0, & k_m < 0 \\ 1, & k_m > 0 \end{cases}, \tag{1}$$

$$b_{m+i} = \begin{cases} 0, & k_m > 0 \\ 1, & k_m < 0 \end{cases}, \qquad w \leq i \leq n-1 \tag{2}$$

And if m is the position of the least non-zero digit of k,

$$b_i = \begin{cases} 1, & i = m \\ 0, & 0 \leq i < m \end{cases}. \tag{3}$$

In this way, at the position of every non-zero digit we can obtain $n - w + 1$ consecutive bits of k except at the position of the least non-zero digit being $m+1$. For the wNAF representation, every non-zero digit (except the least) is followed by at least w zero values. The average number of non-zero digits of k is $(\lfloor \log_2 q \rfloor + 1)/(w+2)$. The average distance between consecutive non-zero digits is $w+2$, i.e. on average $n = w+2$. This means we can obtain 3 consecutive bits on average at the every non-zero digit (except the least). Based on [9] on average we can get 2 least significant bits of the ephemeral key. Thus, on average we can obtain $3(\lfloor \log_2 q \rfloor + 1)/(w+2) - 1$ bits of the ephemeral key k in total. Meanwhile, because the minimal value of n is $w+1$, the minimal length of the consecutive bits is 2. This illustrates that all the sequences of consecutive bits obtained (except the least) are no less than 2 bits.

For the secp256k1 curve implemented in OpenSSL, $\lfloor \log_2 q \rfloor + 1 = 256$, $w = 3$. Also, as introduced previous, the scalar multiplication uses $k+q$ instead of k in most cases, so the total number of bits per signature we obtain is $3(\lfloor \log_2 q \rfloor + 2)/(w+2) - 1 = 153.2$. In theory, two signatures would be enough to recover the 256-bit private key as $2 \times 153.2 = 306.4 > 256$.

3.3 Constructing the Lattice Attack

In this section, we transform the problem of recovering the private key to the HNP instance, and further convert to the CVP/SVP instance in a lattice. Our method is based on the analysis from [39]. But we make some improvements to it. First, the length of the consecutive bits used to construct the lattice is variable while the prior work fixes the length, which may lose some information. Second, in our method the position of consecutive bits is arbitrary in the ephemeral key and does not need to be fixed, while the prior work needs all the consecutive bits at the same position. Finally, from one signature we obtain multiple sequences of consecutive bits, and all of them can be used for constructing the lattice as long as the length of the sequence is satisfied, while the prior work only generates one sequence of consecutive bits for one signature.

To construct an HNP instance using arbitrary consecutive bits, we need to use the following theorem [39]:

Theorem 1. *There exists a polynomial-time algorithm which, given A and B in $[1, q]$, finds $\lambda \in Z_q^*$ such that*

$$|\lambda|_q < B \quad \textit{and} \quad |\lambda A|_q \leq q/B.$$

The value of λ can be computed exploiting the continued fractions.

Recall the ECDSA signature, $s = k^{-1}(h(m) + r \cdot \alpha) \bmod q$. We rewrite it as

$$\alpha r s^{-1} = k - s^{-1}h(m) \mod q. \tag{4}$$

Then assume that we have the l consecutive bits of k with the value of a, starting at some known position j. So k is represented as $k = 2^j a + 2^{l+j} b + c$ for $0 \leq a \leq 2^l - 1$, $0 \leq b \leq q/2^{l+j}$ and $0 \leq c < 2^j$. We apply the theorem with $A = 2^{j+l}$ and $B = q2^{-j-l/2}$, to obtain λ such that

$$|\lambda|_q < q2^{-j-l/2} \quad \text{and} \quad |\lambda 2^{j+l}|_q \leq q/2^{j+l/2}.$$

Plugging the value of k and multiplying by λ, Equation 4 is transformed to

$$\alpha r \lambda s^{-1} = (2^j a - s^{-1}h(m))\lambda + (c\lambda + 2^{l+j} b\lambda) \mod q.$$

Let

$$\begin{cases} t = \lfloor r\lambda s^{-1} \rfloor_q \\ u = \lfloor (2^j a - s^{-1}h(m))\lambda \rfloor_q \end{cases}, \tag{5}$$

where $\lfloor \cdot \rfloor_q$ denotes the reduction modulo q into range $[0, ..., q)$.

We then have that

$$|\alpha t - u|_q < q/2^{(l/2-1)}. \tag{6}$$

This way, we transform to the an HNP instance.

In practice, OpenSSL uses $k + q$ as the ephemeral key. So the Equation 5 remains the same, but the Inequation 6 turns into

$$|\alpha t - u|_q < q/2^{(l/2-\log_2 3)}. \tag{7}$$

Note that, the Equation 7 represents that the $l/2 - \log_2 3 - 1$ most significant bits of $\lfloor \alpha t \rfloor_q$ is u, based on the definition of the HNP. So it should satisfy that $l/2 - \log_2 3 - 1 \geq 1$, i.e. $l > 7$. That means the length of the consecutive bits used to construct the HNP instance should be larger than 7, although we could use all the sequences of the consecutive bits of the ephemeral key in theory.

Next we turn the HNP instance into the lattice problem. We use d triples (t_i, u_i, l_i) to construct a $d+1$ dimensional lattice $L(B)$ spanned by the rows of the following matrix:

$$B = \begin{pmatrix} 2^{l_1+1}q & 0 & \cdots & 0 & 0 \\ 0 & 2^{l_2+1}q & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 2^{l_d+1}q & 0 \\ 2^{l_1+1}t_1 & \cdots & \cdots & 2^{l_d+1}t_d & 1 \end{pmatrix}.$$

Considering the vector $\mathbf{x} = (2^{l_1+1}\alpha t_1 \bmod q, ..., 2^{l_d+1}\alpha t_d \bmod q, \alpha)$, it can be obtained by multiplying the last row vector of B by α and then subtracting appropriate multiples of the first d row vectors. Thus the vector $\mathbf{x}$ belongs to $L(B)$. Let the vector $\mathbf{u} = (2^{l_1+1}u_1, ..., 2^{l_d+1}u_d, 0)$, so the distance between $\mathbf{x}$ and $\mathbf{u}$ is $\|\mathbf{x}-\mathbf{u}\| \le q\sqrt{d+1}$. While the lattice determinant of $L(B)$ is $2^{d+\sum l_i}q^d$, thus the vector $\mathbf{x}$ is very close to the vector $\mathbf{u}$. By solving the CVP problem with the input B and $\mathbf{u}$, the vector $\mathbf{x}$ is revealed, hence the private key α is recovered as it is the last element of the vector $\mathbf{x}$.

To solve the CVP problem in polynomial time, we transform it to an SVP instance. We use d triples (t_i, u_i, l_i) to construct a $d+2$ dimensional lattice $L(B')$ spanned by the rows of the matrix

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

Similarly, the vector $\mathbf{x}' = (2^{l_1+1}(\alpha t_1 - u_1) \bmod q, ..., 2^{l_d+1}(\alpha t_d - u_d) \bmod q, \alpha, -q)$ belongs to the lattice $L(B')$. Its norm satisfies that $\|\mathbf{x}'\| \le q\sqrt{d+2}$, while the lattice determinant of $L(B')$ is $2^{d+\sum l_i}q^{d+1}$. This indicates that the vector $\mathbf{x}'$ is a very short vector. Note that this lattice also contains another vector $(-t_1, ..., -t_d, q, 0) \cdot B = (0, ..., 0, q, 0)$, which is most likely the shortest vector of the lattice. Therefore we expect the second vector in a reduced basis of the lattice is equal to $\mathbf{x}'$ with a "good" chance for a suitably lattice reduction algorithm. Then we acquire the secret key α.

4 Implementation and Experiments

In this section, we apply our method introduced in Section 3 to attack the secp256k1 curve in OpenSSL 1.1.1b. We implement the Flush+Flush attack to obtain the cache side channel information. The details of the implementation and the result of attacking the elliptic curves are provided. Then we implement the lattice attacks and solve them by the BKZ lattice reduction algorithm [49]. The experiments demonstrate the results of our attacks with different parameters. Finally, we compare our method with some previous attacks.

4.1 The Flush+Flush Attack

We launched the cache side channel attack on an Acer Veriton T830 running Ubuntu 16.04. The machine features an Intel Core i7-6700 processor with four execution cores and an 8 MB LLC. The attacking target is the ECDSA implemented in OpenSSL 1.1.1b, which uses wNAF representation in the scalar multiplication. For the experiments, we use the Flush+Flush to attack the 256-bit curve secp256k1.

Get the virtual address. We use a spy process to monitor the `EC_POINT_dbl()`, `EC_POINT_add()` and `EC_POINT_invert()` functions in OpenSSL. So we need to know the virtual addresses of the three functions. First, we get the offsets of the three functions required to monitor in the code of the dynamic library. Then, we use the `mmap()` function to load each page locating the monitored function codes into the virtual address space of the spy process. The `mmap()` function will return the initial address of the page so that we can get the virtual address of the monitored function based on the offset and the address of this page. An alternative way is to use the `dl_iterate_phdr()` function to get the initial address of the dynamic library when loaded into the address space.

To determine the offsets of the memory lines in the dynamic library, we build OpenSSL with debugging symbols. These symbols are not loaded at run time and do not affect the performance of the code. Typically the debugging symbols are not available for attackers, however, they could use reverse engineering [16] to determine the offsets.

Thresholds. We monitor the execution time of the `clflush` instruction to flush the monitored functions. If the time is larger than the threshold, meaning the memory line is accessed by the victim. Otherwise, the memory line is not accessed. The thresholds are calculated for every monitored function. For each address of the monitored functions, we record the time of flushing the cache 1000 times, and take the time larger than 99 percent samples plus 6 cycles as the threshold of this address. The thresholds are recalculated every time before the attack is triggered.

Trigger the monitoring. We monitor the execution time of the address of the three functions. When the time of any one is larger than the corresponding threshold, we start to record execution times, meaning that the attack starts. The record stops when the number of the consecutive execution time of all the monitored addresses less than the threshold is larger than 300. If the number of record is less than 1000, we re-record the execution time. Then we have a valid record. Due to the disturbance of noise, the valid record does not necessarily contain the activities of the monitored functions in the scalar multiplication. So we continuously obtain 10 valid records.

Time slot. For the attack, the spy process divides time into time slots of approximately 2500 cycles. In each slot, the spy flushes the memory lines in the add, double and invert functions (`EC_POINT_dbl()`, `EC_POINT_add()` and `EC_POINT_invert()`) out of the caches. Also, the length of the time slot is chosen to ensure that the three functions only execute once. This allows the spy to correctly distinguish consecutive doubles.

Determine the initial point. In order to precisely recover the sign of the ephemeral key, we need to determine which sample represents the start of the scalar multiplication. We can determine the start of the scalar multiplication by combining the double-add chain and the profiles of the double and add functions with wNAF representation. Also, we can determine the start position by monitoring the code of the copy function in OpenSSL. We note that when the computation of scalar multiplication starts, OpenSSL performs the copy function instead of the add function. Thus we can monitor the copy function, and when the copy function is called in the time slot and the add function is called in the next slot, it means that the scalar multiplication starts.

Experimental results. Fig. 1 shows a fragment of the output of the spy when OpenSSL performs ECDSA with the secp256k1 curve. In this figure, □, ◊ and △ represent "double", "add" and "invert" respectively. From this fragment, three operations are clearly distinguished, so we succeed to obtain the "double-add-invert" chain easily.

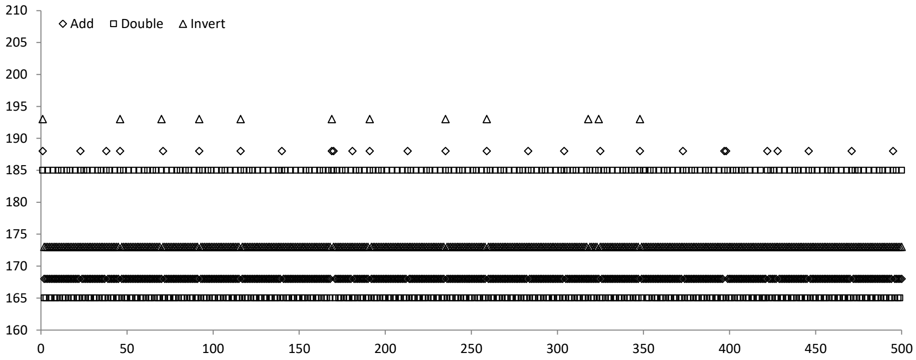

Fig. 1: A fragment of the output of the Flush+Flush attack.

4.2 Lattice Attack

We apply our lattice attack to the curve secp256k1 in our experiments, and we assume that the Flush+Flush attack is perfect, which means we can correctly obtain the "double-add-invert" chain and recover all the information about the digits of the ephemeral key it contains.

The HNP problem can be solved by exploiting both the CVP and SVP instances. However, the CVP problem does not have a polynomial time solution while the SVP problem has. So, with the growth of the dimension of the lattice, the time cost of finding the closest vector grows very fast. Actually, the CVP problem can be converted to the SVP problem, so it is often solved through embedding the CVP into an SVP problem and using the polynomial time solution in practice. Therefore, in our experiments, we only demonstrate the result of using the SVP problem to solve the HNP instance.

We use the BKZ algorithm implemented in `fplll` [52] to solve the SVP problem. When solving an SVP instance, there are two outcomes that either we obtain the private key or a wrong answer. So we denote the success probability as the amount of successfully recovering the private key divided by the total number of the lattice attacks. We want to find the optimal strategy for our attack in terms of the following parameters:

- the minimal value of l (length of the consecutive bits of k)
- the block size of BKZ
- the lattice dimension

Thus we perform a number of experiments with different values of the parameters. In each case, we run 200 experiments and compute the success probability. Because in Section 3 the HNP problem introduced requires $l/2 - \log_2 3 - 1 \geq 1$ to contain more than one bit information, the value of l should be larger than 7. But in our experiments, when $l = 8$, the private key can not be successfully recovered. The reason is that the HNP instance contains not enough information in this case. So the minimal length of the consecutive bits of k is set ranging from 9 to 11 in the experiments. When $l \geq 9$ or $l \geq 10$, the block size in BKZ is set 10 and 20. While $l \geq 11$, the block is set only 10. The number of sequences of the consecutive bits for constructing the lattice denoted as d is ranged from 50 to 230, i.e. the dimension of the lattice for the SVP is $d + 2$.

Table 1 shows the success probability for different dimensions and block sizes of solving the SVP instance in the cases that the minimal length of the consecutive bits ranges from 9 to 11. As shown in the table, we successfully recover the private key of a 256-bit ECDSA only need 60 sequences of consecutive bits with a success probability of 3.0%. These 60 sequences come from up to 60 signatures. Therefore we just need at most 60 signatures to successfully recover the ECDSA private key.

For the fixed minimal length of consecutive bits, increasing the dimension generally increases the probability of success. 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. Also, the higher block sizes perform with a higher success probability, as the stronger reduction allows them to isolate the solution vector better. We believe that the success probability would be surely higher with the block size 30 or the BKZ 2.0 [16].

When l is fixed, to get a suitable success probability, attackers can either increase the dimension or use stronger algorithms (increasing the block size). However, increasing the dimension requires more signatures and enlarging the block size increases the computation time. So attackers can balance the two factors according to their resources and needs.

The success probability also increases as the minimal length of consecutive bits increases. It is because more information is added to the lattice making it easier to search for the desired solution vector. But the increase of the minimal length would lead to requiring more signatures to obtain enough eligible

Table 1: The success probability of solving SVP with different parameters.

Dimension	Block size				
	$l \geq 9$		$l \geq 10$		$l \geq 11$
	10	20	10	20	10
50	0	0	0	0	0
60	0	0	0	0	3.0
70	0	0	0.5	1.0	29.5
80	1.0	1.5	4.0	8.5	49.0
90	0.5	1.0	14.5	22.5	67.0
100	0.5	4.0	21.0	33.5	70.5
110	1.5	3.0	16.0	36.5	79.5
120	2.0	4.5	21.5	35.0	77.5
130	0.5	2.5	24.0	45.5	83.0
140	3.0	8.0	29.0	46.0	88.5
150	3.0	6.0	32.0	49.0	88.0
160	2.5	13.0	27.5	49.0	94.0
170	5.0	12.5	39.0	56.5	93.0
180	3.5	11.5	37.0	57.0	97.0
190	7.0	19.5	39.0	63.5	98.0
200	9.5	26.0	46.0	66.0	99.0
210	10.5	22.0	51.5	66.0	99.5
220	15.0	29.0	51.0	72.5	100.0
230	19.0	30.0	58.5	73.5	99.0

sequences of consecutive bits. Because with the increase of minimal length, the probability of obtaining a longer sequence of consecutive bits becomes lower.

4.3 Comparison with Other Lattice Attacks

Generally, to attack the ECDSA implementation with the wNAF representation, the attackers target the scalar multiplication and use cache side channel attacks to achieve the information about the ephemeral key k. Through side channels, attackers extract a “double-add” chain for the scalar multiplication. However, it is hard to directly recover the whole ephemeral key only from the “double-add” chain. Then, the private key recovery from incomplete information of k is transformed into a problem that can be solved by lattice reduction, such as an HNP or EHNP problem. Finally, the attackers recover the private key by solving the HNP or EHNP problem through being converted to the CVP/SVP problem in the lattice.

We compare the previous attack methods with ours in several aspects, and the result is shown in Table 2. Benger et al. [9] got the least significant bits (LSBs) of the ephemeral key through the Flush+Reload attack although this is not all the information obtained from the side channel. Then they used the LSBs of many signatures to construct an HNP problem and successfully recovered the private key by solving the CVP/SVP instance of a specific lattice converted

Table 2: Comparison with previous attack methods

Methods	Exploited information	HNP or EHNP	# of bits	# of signatures
Benger et al. [9]	LSB	HNP	2	200
Van de Pol et al. [46]	half positions of non-zero digits	HNP	47.6	13
Fan et al. [19]	all positions of non-zero digits	EHNP	105.8	4
Wang et al. [54]	positions of two non-zero digits and the length of the wNAF representation of k	HNP	≥ 2.99	85
Ours	all positions and signs of non-zero digits	HNP	153.2	≤ 60

from the HNP problem. The number of bits extracted from each signature is very small, only an average of 2 bits information can be obtained, thus making it require more than 200 signatures to recover a 256-bit private key with the probability being 3.5%.

Van de Pol et al. [46] improved Benger's attack relying on the property of some specific elliptic curves, that is the order q of the base point is a pseudo-Mersenne prime which can be expressed as $2^n - \epsilon$, where $|\epsilon| < 2^p$, $p \approx n/2$ and n is the length of q. With this property, this attack uses the information from the consecutive non-zero digits whose positions are between $p + 1$ and n extracted from the Flush+Reload attack, not only the LSBs. It is able to extract 47.6 bits per signature on average for the secp256k1 curve and recover the private key with 13 signatures. Then they constructed an HNP instance using this information and successfully recovered the private key. Although this work greatly reduced the number of signatures needed to recover the private key, the information from the cache side channel is not fully utilized. But our work can use all the information from the cache side channel. Moreover, their work is only effective to some special curves, but ours can be applied to all the curves without any restriction.

Fan et al. [19] extracted all positions of digits from the Flush+Reload attack and took advantage of them to construct an EHNP instance. They managed to obtain on average 105.8 bits per signature for the secp256k1 curve and only need 4 signatures to recover the private key with the probability being 8%. Compared with their work, ours obtains more information about the ephemeral key, i.e. the sign of all non-zero digits, from the side channel by analysing the OpenSSL implementation. We can extract 153.2 bits information per signature. Although it uses more signatures to recover the private key in our lattice attack, theoretically, the number of signatures needed is less than Fan's if a more suitable lattice is constructed.

Wang et al. [54] obtained the positions of two non-zero digits and the length of the wNAF representation of k from the Flush+Reload attack. They could obtain no less than 2.99 bits information per signature and exploit the HNP to recover the private key for the 256-bit curve using 85 signatures. Obviously, our method obtains more information and uses fewer signatures to recover the private key.

In summary, our method exploits both the signs and the positions of the non-zero digits of the ephemeral key k achieved from the Flush+Flush attack. It extracts the largest amount of information, on average 153.2 bits per signature. In theory, the number of signatures needed to recover the private key is only 2. However, due to the limit of our method of lattice construction, we need at most 60 signatures, which is more than Fan's and Van de Pol's. One of our future work is to find an efficient lattice construction to solve the problem (e.g., an EHNP-based solution).

5 Discussion

5.1 Scalar Multiplication in Other Cryptographic Libraries

Besides the OpenSSL library, there are many cryptographic libraries that implement the scalar multiplication. But not all of them use the wNAF representation. We introduce some implementations of the scalar multiplication in some popular cryptographic libraries.

The mbed TLS [3] is an open source SSL library licensed by ARM Limited. It does not use the wNAF algorithm to represent the scalar. It uses a comb method that generates a sequence of bit-strings to represent the scalar k so that every bit-string represents an odd number based on modifying the method in [25]. When computing the scalar multiplication, it traverses the sequence and directly indexes the value in the precomputed points to perform the addition. It does not use the invert function, and this representation is hard to attack by the cache side channels because all the elements of the sequence are not zero.

Libgcrypt [1] is a cryptography library developed as a separated module of GnuPG. It directly uses the binary representation of the scalar instead of the wNAF representation to compute the scalar multiplication in the prime field. To resist the side channel attacks, it performs an extra addition operation when the binary bit of the scalar is zero so that in every bit the double and addition are both performed. Therefore, lattice attacks to this library are invalid.

5.2 The Method of Constructing Lattice Attacks

In our work, we construct an HNP instance using partial information about the k obtained from the cache side channel attack. From the Inequation 7, the length of the most significant bits of $\lfloor \alpha t_i \rfloor$ is $l/2 - \log_2 3 - 1$. When constructing the lattice, it should contain the information about the most significant bits of $\lfloor \alpha t_i \rfloor$. Thus it requests $l/2 - \log_2 3 - 1 \geq 1$, so l representing the minimal length of the consecutive bits needs to be larger than 7. In practical, only when $l > 8$, we successfully recover the ECDSA private key in our experiments. However, the average length of the consecutive bits is 3, which is much smaller than needed. This makes it impossible to use all the bits obtained through the cache side channel attack to construct the lattice attack. Although in theory we only need 2 signatures to recover the 256-bit private key, in practical we need at least 60 signatures to get enough information satisfied the restriction on l to construct the lattice attack. Therefore due to the restriction on l, we can not make full use of the information obtained from the cache side channel although we obtained 153.2 bits information per signature on average.

However, based on the work in [39] and [33], in theory when the length of the consecutive bits is 2, this sequence of the consecutive bits can be used to construct the lattice attack. In our work, all the sequences of consecutive bits (except the least) obtained from the cache side channel satisfy $l \geq 2$, and the length the consecutive least significant bits is larger than 1. That means at least 152.2 bits information we obtained is valid and exploitable, only 1 bit information may be discarded. Therefore, if we find an efficient way to construct the lattice attack, making full use of all the information of the consecutive bits, the number of signatures required to recover the private key can be greatly reduced, even the best case only 2 signatures is enough for a 256-bit private key. In theory, this result is better than the prior work in [19] which theoretically needs 3 signatures for a 256-bit private key. This is what we are going to do next, finding a suitable and powerful method to construct the lattice attack.

6 Related work

6.1 Partial Nonce Disclosure Attacks on DSA/ECDSA

Several works have proposed different attacks that exploit partial nonce disclosure to recover long-term private keys of the algorithms based on the discrete logarithm problem. Some of them focus on how to exploit the partial information efficiently to construct some appropriate lattice attacks to recover the private key. And others pay more attention to how to efficiently obtain the information about the ephemeral key.

Boneh and Venkatesan [11] initially investigated to use the partial information of the ephemeral key to construct an HNP problem and recovered the private key of Diffie-Hellman by solving it using the lattice reduction algorithm. Howgrave-Graham and Smart [26] extended the work of Boneh and Venkatesan [11] and showed how to recover the DSA private key by constructing an HNP instance from leaked LSBs and MSBs of the ephemeral key. Nguyen and Shparlinski [39] improved their method and gave a provable polynomial-time attack that knowing the $l \geq 3$ LSBs, the $l+1$ MSBs or any $2l$ consecutive bits of a certain number of ephemeral keys was enough for recovering the DSA private key. They recovered a 160-bit DSA key only using the 3 LSBs of a certain number of ephemeral keys. Further, they extended these results to ECDSA [40]. Liu and Nguyen [33] improved the results that only 2 LSBs were required for breaking a 160-bit DSA key. In 2014, Benger et al. [9] extended the technique in [39] to use a different length of leaked LSBs for each signature. They recovered the secret key of OpenSSL's ECDSA implementation for the curve secp256k1 using about 200 signatures. Van de Pol et al. [46] exploited the property of the modulus in some elliptic curves so that they could use all of the information leaked in the top half of the ephemeral keys to construct the HNP instance, allowing them to recover the secret key after observing only 14 signatures. Fan et al. [19] transformed the problem of recovering the secret key to the extended hidden number problem (EHNP) which was solved by the lattice reduction algorithm. Then the number of signatures needed was reduced to 4.

Brumley and Hakala [13] used an L1 data cache-timing attack to recover the LSBs of ECDSA ephemeral keys in OpenSSL 0.9.8k. They recovered a 160-bit ECDSA private key using the attack in [26] with collecting $2,600$ signatures ($8K$ with noise). Analogously, Acıiçmez et al. [5] used an L1 instruction cache-timing attack to recover the LSBs of DSA ephemeral keys in OpenSSL 0.9.8l. It required $2,400$ signatures ($17K$ with noise) to recover a 160-bit DSA private key. Besides, both attacks required HyperThreading architectures. In 2011, Brumley and Tuveri [14] mounted a remote timing attack on the implementation of ECDSA with binary curves in OpenSSL 0.9.8o. They obtained the MSBs of the ephemeral keys through the timing attack and recovered the private key after collecting information over 8,000 TLS handshakes. In 2013, Mulder et al. [17] took advantage of a template attack to recover some LSBs of the ephemeral key. They improved the Bleichenbacher's FFT-based (Fast Fourier Transform) attack by using BKZ for range reduction to solve the HNP problem. Their attack could extract the entire private key using a 5-bit leak of the ephemeral key from 4 000 signatures. Benger et al. [9] and Van de Pol et al. [46] used the Flush+Reload technique to acquire the information of the ephemeral key. Allan et al. [7] improved the results in [46] by using a performance-degradation attack to amplify the side-channel. The amplification allowed them to recover a 256-bit private key in OpenSSL 1.0.2a after observing only 6 signatures. Genkin et al. [20] performed electromagnetic and power analysis attacks on mobile phones. They showed how to construct HNP triples when the signature uses the low s-value. Pereida et al. [45] showed that the DSA implementation in OpenSSL is vulnerable to cache-timing attacks due to a programming error, and exploited the vulnerability to attack against SSH (via OpenSSH) and TLS (via stunnel). In 2017 Zhang et al. [57] extended the attack in [40] to SM2 Digital Signature Algorithm (SM2-DSA), which is a Chinese version of ECDSA.

6.2 Cache Side Channel Attacks

In 2002, Page [43] first described a theoretical cache based side channel attack on the collection of cache profiles for a large amount of chosen plaintexts. Later Tsunoo et al. [53] proposed the first practical implementation of the cache attack on the DES cryptographic algorithm. The first practical cache side-channel attack on AES was appeared in 2005 by Bernstein [10], and it statically analyzed the relation between the overall execution time and lookup table indexes affected by the cache behavior. Bonneau and Mironov [12] showed how to exploit cache collisions when indexing the AES lookup tables to recover the AES secret key.

In 2006, Osvik et al. [42] presented two access-driven attacks: Evict+Prime and Prime+Probe. Although the latter one proved to be significantly more efficient, both of them recovered the AES encryption key used by an OpenSSL server. Acıiçmez and Koc [6] presented a trace-driven attack targeting AES that exploited internal table lookup collisions in the cache during the first round. In 2011, Gullasch et al. [24] first presented a new attack which was later called the Flush+Reload is used to attack AES encryption by blocking the execution of AES after each memory access using the process scheduling algorithm. In 2014, Irazoqui [28] exploit the Flush+Reload technique to attack the AES.

Percival [44] presented the first cache attack on RSA using the data cache collision. Acıiçmez in [4] first exploited that the instruction cache leaked information when performing RSA encryption. Brumley and Hakala [13] mounted an L1 data cache-timing attack to recover the LSBs of ECDSA ephemeral keys from the `dgst` command line tool in OpenSSL 0.9.8k and then used a lattice attack to recover a 160-bit ECDSA private key. While Acıiçmez et al. [5] used an L1 instruction cache-timing attack to recover the LSBs of DSA ephemeral keys from the same tool in OpenSSL 0.9.8l to recover a 160-bit DSA private key. In 2014 Yarom et al. proposed the Flush+Reload attack on RSA using the L3 cache [56]. Then, again Yarom et al. used the Flush+Reload technique to recover the secret key from an ECDSA signature algorithm [55].

Cloud computing systems also can be attacked using the cache side channel attacks. In 2009, Ristenpart et al. [48] presented the methods to co-locate an attackers virtual machine (VM) with a potential victims VM with a success probability of 40% in the Amazon EC2 cloud. Also, the authors showed that the cache side-channel attacks are both practical and applicable to real world scenarios by successfully recovering keystrokes from the co-resident victims VM. This work provided a practical method to find the co-resident VMs, which was the basis of the cache attacks cross VMs because the caches were only shared with co-resident VMs. For instance, Flush+Reload was used to attack AES [28] and RSA [56] in the cross-VM scenario. Also, the Prime+Probe method was also adapted to work in virtualized environments by Zhang et al. [58] and Liu [32] to recover ElGamal encryption keys. Irazoqui et al. [27] recovered AES keys in virtualized environments with Bernstein's attack. Benger et al. [9] and Van de Pol et al. [46] demonstrated the viability of the Flush+Reload technique to recover ECDSA encryption keys. Fan et al. [19] proposed a new way of extracting and utilizing information from the Flush+Reload attack. Flush+Flush [23] and Prime+Abort [18] are variants of Flush+Reload. They exploited different methods to observe the cache access patterns to recover the private key and could be used in both local and virtualized environments.

7 Conclusion and Future Work

In this paper, we demonstrate a practical attack on the ECDSA algorithm implemented in OpenSSL with the scalar multiplication using the wNAF representation. We improve the original cache side channel attack by adding an extra monitor to the invert function and get the extra information about the signs of all non-zero digits of the ephemeral key. Then we exploit a new method to get the consecutive bits of the ephemeral key at each position of the non-zero digits through the "double-add-invert" chain obtained by the cache side channel attack. From the side channel, we obtain 153.2 bits of information per signature for the 256-bit ECDSA secp256k1 curve. We construct a lattice attack using the HNP problem to recover the ECDSA private key. This is the first work to obtain the information about the signs of the non-zero digits of the ephemeral key and make use of it to recover the ECDSA private key. We implement our method to attack the secp256k1 curve. The experiments show that we successfully recover the private key from only 60 signatures.

In the future, we will optimize this work to use fewer signatures and achieve higher success probability by finding ways to decrease the length of consecutive bits required and using more efficient lattice reduction algorithms such as the BKZ2.0. Next, we will try to find a more efficient lattice construction to recover the private key, making full use of the information achieved (e.g., an EHNP-based solution). Also, we will extend this attack to other cryptographic engines, and also the blinded ephemeral keys of ECDSA.

References

1. Libgcrypt. https://www.gnupg.org/software/libgcrypt/index.html
2. OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/
3. SSL Library mbed TLS / PolarSSL. https://tls.mbed.org/
4. Acıiçmez, O.: Yet another microarchitectural attack: Exploiting I-cache. In: Proceedings of the 2007 ACM workshop on Computer Security Architecture, CSAW. pp. 11–18. ACM (2007)
5. Acıiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Cryptographic Hardware and Embedded Systems, CHES 2010. pp. 110–124. Springer Berlin Heidelberg (2010)
6. Acıiçmez, O., Koç, Ç.K.: Trace-driven cache attacks on AES (short paper). In: Information and Communications Security. pp. 112–121. Springer Berlin Heidelberg (2006)
7. Allan, T., Brumley, B.B., Falkner, K., van de Pol, J., Yarom, Y.: Amplifying side channels through performance degradation. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. pp. 422–435. ACSAC '16, ACM (2016)
8. American National Standards Institute: ANSI X9.62-2005, Public Key Cryptography for the Financial Services Industry : The Elliptic Curve Digital Signature Algorithm (ECDSA) (2005)
9. Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: "ooh aah... just a little bit" : A small amount of side channel can go a long way. In: Cryptographic Hardware and Embedded Systems – CHES 2014. pp. 75–92. Springer Berlin Heidelberg (2014)
10. Bernstein, D.J.: Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf (2005)
11. Boneh, D., Venkatesan, R.: Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In: Advances in Cryptology — CRYPTO '96. pp. 129–142. Springer Berlin Heidelberg (1996)

12. Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Cryptographic Hardware and Embedded Systems - CHES 2006. pp. 201–215. Springer Berlin Heidelberg (2006)
13. Brumley, B.B., Hakala, R.M.: Cache-timing template attacks. In: Advances in Cryptology - ASIACRYPT 2009 - 15th International Conference on the Theory and Application of Cryptology and Information Security. pp. 667–684. Springer Berlin Heidelberg (2009)
14. Brumley, B.B., Tuveri, N.: Remote timing attacks are still practical. In: Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security. pp. 355–371. Springer Berlin Heidelberg (2011)
15. Cao, W., Feng, J., Chen, H., Zhu, S., Wu, W., Han, X., Zheng, X.: Two lattice-based differential fault attacks against ECDSA with wNAF algorithm. In: Information Security and Cryptology - ICISC 2015 - 18th International Conference. pp. 297–313. Springer International Publishing (2015)
16. Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better lattice security estimates. In: Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security. pp. 1–20. Springer Berlin Heidelberg (2011)
17. De Mulder, E., Hutter, M., Marson, M.E., Pearson, P.: Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In: Cryptographic Hardware and Embedded Systems - CHES 2013. pp. 435–452. Springer Berlin Heidelberg (2013)
18. Disselkoen, C., Kohlbrenner, D., Porter, L., Tullsen, D.: Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In: Proceedings of the 26th USENIX Conference on Security Symposium. pp. 51–67. SEC'17, USENIX Association (2017)
19. Fan, S., Wang, W., Cheng, Q.: Attacking OpenSSL implementation of ECDSA with a few signatures. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1505–1515. CCS '16, ACM (2016)
20. Genkin, D., Pachmanov, L., Pipman, I., Tromer, E., Yarom, Y.: ECDSA key extraction from mobile devices via nonintrusive physical side channels. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1626–1638. CCS '16, ACM (2016)
21. Gordon, D.M.: A survey of fast exponentiation methods. Journal of Algorithms **27**(1), 129 – 146 (1998)
22. Goudarzi, D., Rivain, M., Vergnaud, D.: Lattice attacks against elliptic-curve signatures with blinded scalar multiplication. In: Selected Areas in Cryptography – SAC 2016. pp. 120–139. Springer International Publishing (2017)
23. Gruss, D., Maurice, C., Wagner, K., Mangard, S.: Flush+ Flush: A fast and stealthy cache attack. In: Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 279–299. Springer International Publishing (2016)
24. Gullasch, D., Bangerter, E., Krenn, S.: Cache games – bringing access-based cache attacks on AES to practice. In: 32nd IEEE Symposium on Security and Privacy, S&P 2011. pp. 490–505 (May 2011)
25. Hedabou, M., Pinel, P., Bénéteau, L.: A comb method to render ECC resistant against side channel attacks. IACR Cryptology ePrint Archive **2004**, 342 (2004)
26. Howgrave-Graham, N.A., Smart, N.P.: Lattice attacks on digital signature schemes. Designs, Codes and Cryptography **23**(3), 283–290 (Aug 2001)
27. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Fine grain Cross-VM attacks on Xen and VMware. In: 2014 IEEE Fourth International Conference on Big Data and Cloud Computing. pp. 737–744 (Dec 2014)

28. Irazoqui, G., Inci, M.S., Eisenbarth, T., Sunar, B.: Wait a minute! a fast, cross-VM attack on AES. In: Research in Attacks, Intrusions and Defenses. pp. 299–319. Springer International Publishing (2014)
29. J. Callas, L. Donnerhacke, H. Finney, D. Shaw and R. Thayer: OpenPGP Message Format (RFC 4880) (2007)
30. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security **1**(1), 36–63 (Aug 2001)
31. Koyama, K., Tsuruoka, Y.: Speeding up elliptic cryptosystems by using a signed binary window method. In: Advances in Cryptology — CRYPTO' 92. pp. 345–357. Springer Berlin Heidelberg (1992)
32. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. pp. 605–622. IEEE Computer Society (2015)
33. Liu, M., Nguyen, P.Q.: Solving BDD by enumeration: An update. In: Topics in Cryptology - CT-RSA 2013 - The Cryptographers' Track at the RSA Conference 2013. pp. 293–309. Springer Berlin Heidelberg (2013)
34. Miyaji, A., Ono, T., Cohen, H.: Efficient elliptic curve exponentiation. In: Proceedings of the First International Conference on Information and Communication Security. pp. 282–291. ICICS '97, Springer-Verlag (1997)
35. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008), `https://bitcoin.org/bitcoin.pdf`
36. National Institute of Standards and Technology: FIPS PUB 186-4 Digital Signature Standard (DSS) (July 19 2013)
37. Nguyen, P.: Cryptanalysis of the goldreich-goldwasser-halevi cryptosystem from crypto '97. In: Advances in Cryptology — CRYPTO' 99. pp. 288–304. Springer Berlin Heidelberg (1999)
38. Nguyen, P.Q.: The dark side of the hidden number problem: Lattice attacks on DSA. In: Cryptography and Computational Number Theory. pp. 321–330. Birkhäuser Basel (2001)
39. Nguyen, P.Q., Shparlinski, I.E.: The insecurity of the digital signature algorithm with partially known nonces. Journal of Cryptology **15**(3), 151–176 (Jun 2002)
40. 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 (Sep 2003)
41. Nguyen, P.Q., Stern, J.: Lattice reduction in cryptology: An update. In: Algorithmic Number Theory. pp. 85–112. Springer Berlin Heidelberg (2000)
42. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: The case of AES. In: Topics in Cryptology - CT-RSA 2006 - The Cryptographers' Track at the RSA Conference 2006. pp. 1–20. Springer Berlin Heidelberg (2006)
43. Page, D.: Theoretical use of cache memory as a cryptanalytic side-channel. IACR Cryptology ePrint Archive **2002**, 169 (2002)
44. Percival, C.: Cache missing for fun and profit (2005)
45. Pereida García, C., Brumley, B.B., Yarom, Y.: Make sure DSA signing exponentiations really are constant-time. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1639–1650. CCS '16, ACM (2016)
46. van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: Topics in Cryptology - CT-RSA 2015 - The Cryptographers' Track at the RSA Conference 2015. pp. 3–21. Springer International Publishing (2015)

47. Rankl, W.: Smart card applications: Design models for using and programming smart cards (01 2007)
48. Ristenpart, T., Tromer, E., Shacham, H., Savage, S.: Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 199–212. CCS '09, ACM (2009)
49. Schnorr, C.P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Mathematical Programming **66**(1), 181–199 (Aug 1994)
50. Solinas, J.A.: Efficient arithmetic on koblitz curves. Designs, Codes and Cryptography **19**(2), 195–249 (Mar 2000)
51. T. Dierks and E. Rescorla: The Transport Layer Security (TLS) Protocol Version 1.2 (RFC 5246) (2008)
52. The FPLLL development team: fplll, a lattice reduction library (2016), https://github.com/fplll/fplll, available at https://github.com/fplll/fplll
53. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H.: Cryptanalysis of DES implemented on computers with cache. In: Cryptographic Hardware and Embedded Systems - CHES 2003. pp. 62–76. Springer Berlin Heidelberg (2003)
54. Wang, W., Fan, S.: Attacking OpenSSL ECDSA with a small amount of side-channel information. Science China Information Sciences **61**(3), 032105 (Aug 2017)
55. Yarom, Y., Benger, N.: Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. IACR Cryptology ePrint Archive **2014**, 140 (2014)
56. Yarom, Y., Falkner, K.: Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In: Proceedings of the 23rd USENIX Conference on Security Symposium. pp. 719–732. SEC'14, USENIX Association (2014)
57. Zhang, K., Xu, S., Gu, D., Gu, H., Liu, J., Guo, Z., Liu, R., Liu, L., Hu, X.: Practical partial-nonce-exposure attack on ECC algorithm. In: 2017 13th International Conference on Computational Intelligence and Security (CIS). pp. 248–252 (Dec 2017)
58. Zhang, Y., Juels, A., Reiter, M.K., Ristenpart, T.: Cross-VM side channels and their use to extract private keys. In: the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012. pp. 305–316. ACM (2012)