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

Ben Trovato
trovato@corporation.com

ABSTRACT

This paper presents a practical lattice attack on ECDSA implementations which use the windowed Non-Adjacent-Form (wNAF) representation 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 the only ones exploited in the literature. By monitoring the invert function, we obtain the signs of the wNAF representation 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, 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) and Extended Hidden Number Problem (EHNP) respectively, which are 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 and EHNP instances. The experimental results show that the private key is successfully recovered from 2 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 the least number of signatures required for ECDSA private key recovery.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Woodstock '18, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

KEYWORDS

ECDSA , windowed Non-Adjacent-Form , OpenSSL , Lattice Attack , Hidden Number Problem , Cache Side Channel

ACM Reference Format:
Ben Trovato. 2018. Another Lattice Attack against the wNAF Implementation of ECDSA to Recover More Bits per Signature. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

The Elliptic Curve Digital Signature Algorithm (ECDSA) [7, 28] is a digital signature scheme based on the elliptic-curve cryptography (ECC), widely used in many popular applications, such as TLS [47], OpenPGP [27], smart card [43], and Bitcoin [34]. The core operation of ECDSA sign/verify is the scalar multiplication of a base point (or generator) over elliptic curves by a random nonce (or ephemeral key). And the security of ECDSA relies on the intractability of the elliptic curve discrete logarithm problem (ECDLP) that the inability to compute the scalar (ephemeral key) given the original (base) and product points.

However, in actual implementations, the ephemeral key and the scalar multiplication need careful protection. Side channel attacks can obtain the information on the ephemeral key during the scalar multiplication. Furthermore, as long as some bits of the ephemeral key are leaked, the ECDSA private key will be recovered [24, 36–38].

The partial ephemeral key exposure attack was first proposed by Howgrave-Graham and Smart [24] in 2001 to recover the whole DSA private key. It was improved by Nguyen and Shparlinski [37] and extended to ECDSA [38]. The basic idea is to construct a Hidden Number Problem (HNP) with the partial knowledge of the ephemeral keys, and then reduce to the Closest Vector Problem (CVP) or the Shortest Vector Problem (SVP) in a lattice to calculate the private key. In 2007, the Extended Hidden Number Problem (EHNP) [23] was introduced as another way to construct the problem to be solved in the lattice to recover the private key using the partial ephemeral key information.

While the cache side channel attack is an efficient way to obtain the partial information of the ephemeral key. With the Flush+Reload cache side channels [53], attackers can get a "double-add" chain and extract some useful information about the ephemeral key. Benger et al. [8] extracted the LSBs of the ephemeral keys from the chain and constructed an HNP instance to recover the ECDSA private key from about 200 signatures. In 2015, Van de Pol et al. [50] improved Benger's attack, which exploited the positions of higher half

non-zero digits of the wNAF representations of the ephemeral key extracted from the "double-add" chain. In 2016, Fan et al. [17] extracted and utilized the information from the the "double-add" chain to construct the EHNP to recover the private key of ECDSA. The number of signatures needed is reduced to 4. In 2017, Wang et al. [51] 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 [8], half non-zero digits [50], all non-zero digits [17] or two non-zero digits with the total length [51]. In summary, all information exploited in these attacks are from the "double-add" chain, and no more extra information about the ephemeral key is extracted from the side channels.

In our paper, we focus on ECDSA with the wNAF representation of the ephemeral key. We propose new methods 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. This function inverts a number so that the subtraction is replaced by addition and the space storing the precomputed points is reduced by half. When computing the scalar multiplication, the invert function is called if the sign of current non-zero digit of the ephemeral key is opposite to the previous one, and the absolute value of this digit is used to index the precomputed points. Otherwise, the invert function is not called. This fact helps us to determine the sign of the non-zero digits of the wNAF representation. Therefore, we add a extra monitor to the invert function so that we get a Double-Add-Invert chain through the cache side channel and can extract more information about the ephemeral key compared to previous attacks. After that, we try to use all the information achieved to recover the ECDSA private key. we first recover the consecutive bits at the position of every non-zero digits of the ephemeral key based on the Double-Add-Invert chain and construct an HNP instance to recover the ECDSA private key. Due to the length limitation of the consecutive bits, this method needs more signatures than [50]. Then, we propose a novel method to construct an EHNP [23] instance to recover the ECDSA private key by converting it to the SVP problem and solving it with lattice reduction algorithms. In the novel method, the short vectors are much easier to be found by the lattice reduction algorithm comparing to the constructed lattice in [17]. And the signatures needed to recover the ECDSA private key are significantly reduced, which can attain the optimal bound(i.e. 2 for the 256-bit curve).

We applied our attack methods to the secp256k1 curve in OpenSSL 1.1.1b. We choose the Flush+Flush [21] attack to monitor the functions of double, add and invert. From the Double-Add-Invert chain obtained 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. We extract on average 156.2 bits of information from the Double-Add-Invert Chain per signature for the 256-bit elliptic curve. If the Double-Add-Invert chain is perfect, using the HNP to recover the ECDSA private key, we need about 60 signatures to recover the private key with a probability of 3%. While using the EHNP, with some new techniques introduced to improve the success probability, 3 signatures is enough to recover the secret key with a high success probability no less than 73%. And when we have only two signatures, we can recover the secret key with a success probability no less than 24.3%. This result attains the optimal bound and is the best one as ever known.

We have improved the attacks against ECDSA in two aspects. And our contributions are summarized as follows:

- We improve the cache side channel attacks by monitoring the invert function to obtain the sign information of the ephemeral key. It is the first work to exploit all signs of the non-zero digits of the ephemeral key in side channel attacks.
- We present new lattice attacks to recover the private key of ECDSA with the wNAF representation. First, We extract multiple consecutive bits of the ephemeral key by converting the wNAF representation to the binary representation and construct the HNP instance to recover the ECDSA private key. Second, We propose a novel method to construct an EHNP instance to recover the ECDSA private key which is better than [17].
- We apply our methods to attack the secp256k1 curve in OpenSSL 1.1.1b. Through the cache side channel, 156.2 bits of information per signature are obtained on average. The experiments show that only 2 signatures are enough to recover the private key with a success probability no less than 24.3%.

The rest of this paper is organized as follows. Section 2 presents some preliminaries. Section 3 shows how to improve the cache side channel to obtain more information. Section 4 and 5 show how to set the lattice attacks with HNP and EHNP problems, respectively. Section 6 compares different lattice attacks. Section 7 contains some extended discussions. Section 8 introduces some related works. Section 9 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 the extended hidden number

problem are introduced to dispose the data obtained from cache side channels.

2.1 The Elliptic Curve Digital Signature Algorithm

ECDSA [7, 28] is the migration of the Digital Signature Algorithm (DSA) [35] 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:

(1) Select a random integer as the 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

Scalar multiplication kG on the elliptic curve means that the point G is added to itself k times, where G is a point defined on the elliptic curve over the finite field, and the scalar k is a big integer. There are several algorithms to implement the scalar multiplication. In OpenSSL, scalar multiplication in the prime field is implemented using the wNAF representation [19, 29, 33, 46] 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 satisfying $-2^w < k_i < 2^w$, 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.

From the Algorithm 2, 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.

Algorithm 1 Conversion to wNAF Representation

Input: Scalar k, window size w
Output: k in wNAF: k_0, k_1, k_2, ...
1: $i \leftarrow 0$
2: **while** $k > 0$ **do**
3: **if** $k \bmod 2 = 1$ **then**
4: $k_i \leftarrow k \bmod 2^{w+1}$
5: **if** $k_i \geq 2^w$ **then**
6: $k_i \leftarrow k_i - 2^{w+1}$
7: **end if**
8: $k \leftarrow k - k_i$
9: **else**
10: $k_i \leftarrow 0$
11: **end if**
12: $k \leftarrow k/2$
13: $i \leftarrow i + 1$
14: **end while**

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**

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 [13]. In most cases, the multiplication is done as $(k + q)G$, but it can still be vulnerable in lattice attacks [20].

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 [1], 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.

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

```
1: r ← 0, is_neg ← 0, r_is_inverted ← 0
2: for i from l − 1 to 0 do
3:     if r ≠ 0 then
4:         EC_POINT_dbl(r)   // double
5:     end if
6:     if k_i ≠ 0 then
7:         is_neg ← (k_i < 0)
8:         if is_neg then
9:             k_i ← −k_i
10:        end if
11:        if is_neg ≠ r_is_inverted then
12:            if r ≠ 0 then
13:                EC_POINT_invert(r)   // invert
14:            end if
15:            r_is_inverted ←!r_is_inverted
16:        end if
17:        if r = 0 then
18:            r ← EC_POINT_copy(k_iG)
19:        else
20:            r ← EC_POINT_add(r, k_iG)   // add
21:        end if
22:    end if
23: end for
24: return r
```

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.

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 [9, 11, 22, 39, 53] since it is demonstrated feasible in theory [40]. We introduce two typical access-driven attacks called Flush+Reload [53] and Flush+Flush [21] that can be used to monitor the functions of the implementation of ECDSA.

The Flush+Reload attack [53] 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:** 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:** the attacker waits a moment while the victim is running.
- **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. The longer reload time 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 [21] also assumes the 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 and the third phase are merged together in the Flush+Flush attack. Therefore the Flush+Flush attack is more accurate than the Flush+Reload attack.

2.5 The Hidden Number Problem and Lattice Attack

For the wNAF algorithm, the if-then block (Line 4 in Algorithm 2) 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. But this is not enough to determine the complete ephemeral key, only some bits of the ephemeral key are obtained. The attacker then use this information to launch a lattice attack to recover the private key.

The Hidden Number Problem (HNP) is first presented by Boneh and Venkatesan [10] in 1996. It is used to recover the secret key of Diffie-Hellman key exchange [10], DSA [24] and ECDSA [38], 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

$$v_i = |\alpha t_i - u_i|_q \leq q/2^{l+1}, \quad 1 \leq i \leq d,$$

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 Extended Hidden Number Problem (EHNP) introduced in [23] is stated as follows, which also can be used to recover the ECDSA private key [17]. Let N be a prime number. Given u congruences

$$\beta_i x + \sum_{j=1}^{l_i} a_{i,j} k_{i,j} \equiv c_i \mod N, \quad 1 \leq i \leq u \ ,$$

where $k_{i,j}$ and x are unknown variables satisfying $0 \leq k_{i,j} \leq 2^{\varepsilon_{i,j}}$ and $0 < x < N, \beta_i, a_{i,j}, c_i, l_i$ and $\varepsilon_{i,j}$ are all known. The EHNP is to find the unknown x satisfying the conditions above.

For ECDSA algorithm, attackers take advantage of the form of equation $s = k^{-1}(h(m) + r \cdot \alpha) \bmod q$. Based on the partial information about the ephemeral key obtained by the attackers through the cache side channels, they transform this equation to satisfy the form of HNP or EHNP. Then the private key as the hidden number can be recovered by solving the SVP/CVP problem in lattice converted from HNP or EHNP using the lattice reduction algorithm such as LLL [30] or BKZ [45].

3 IMPROVING CACHE SIDE CHANNEL ATTACK ON WNAF REPRESENTATION

This section proposes how to achieve more information about the ephemeral key with the wNAF representation through the cache side channel. First, we analyse the invert function in the scalar multiplication with wNAF representation in the ECDSA algorithm and show how to use the Double-Add-Invert chain from the cache side channel to extract the information about the ephemeral key. Then we implemented the Flush+Flush attack to the secp256k1 curve in OpenSSL 1.1.1b to obtain the cache side channel information. The details of the implementation and the result of attacking the elliptic curves are provided.

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. 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 in 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 their signs.

3.2 The Implementation of Flush+Flush Attack

We launched the cache side channel attack to obtain the partial information of the ECDSA 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. Both the Flush+Reload and Flush+Flush attacks can be used, but we choose the Flush+Flush to attack the 256-bit curve secp256k1 for the experiments due to the accuracy as described in Section 2.4.

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.

Threshold. 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.

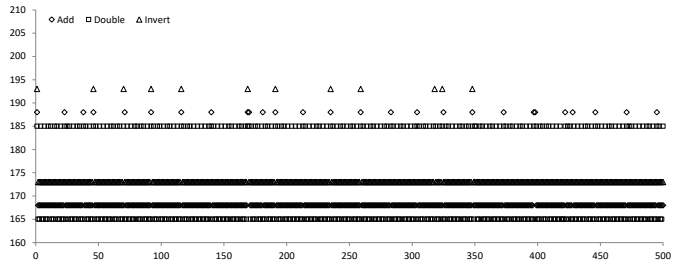

Figure 1: A fragment of the output of the Flush+Flush attack.

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, $\square$, $\Diamond$ and $\triangle$ 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.

4 RECOVER THE ECDSA PRIVATE KEY WITH HNP

We first recover the consecutive bits at the position of every non-zero digits for the ephemeral key k, exploiting the information obtained from the side channel. Then we translate

the problem of recovering the ECDSA private key to the HNP problem and work it out by solving the approximate CVP/SVP Problem with the lattice reduction algorithm.

We applied our attack to the secp256k1 curve. We measured the success probability of recovering the ECDSA private key in different minimal values of l (length of the consecutive bits of k), block sizes of BKZ and lattice dimensions. Due to the limitation of l, we need at least 247.45 signatures to recover the ECDSA private key using the HNP problem.

4.1 Recovering Consecutive Bits

First, we denote the wNAF representation of k as $k = \sum k_i 2^i$, and the binary representation as $k = \sum b_i 2^i$. 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 $0 \le i < j$. But for arbitrary non-zero digits, it can not directly 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:

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

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

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

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

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 one). Based on [8] 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 $2 \times 153.2 = 306.4 > 256$.

4.2 Constructing the Lattice Attack with HNP

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 [37]. 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[37]:

THEOREM 4.1. *There exists a polynomial-time algorithm which, given A and B in $[1, q]$, finds $\lambda \in Z_q^*$ such that*

$$|\lambda|_q < B \quad \text{and} \quad |\lambda A|_q \le q/B.$$

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 $0 \le a \le 2^l - 1$, $0 \le b \le q/2^{l+j}$ and $0 \le c < 2^j$. We apply the theorem with $A = 2^{j+l}$ and $B = q2^{-j-l/2}$, to obtain λ such that

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

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 an HNP instance.

In practice, OpenSSL uses $k + q$ as the ephemeral key. So the Equation 5 remains the same, but the Inequality 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 $l/2 - \log_2 3 - 1 \ge 1$, 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 $\|\mathbf{x} - \mathbf{u}\| \leq q\sqrt{d+1}$. 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 $\|\mathbf{x}'\| \leq q\sqrt{d+2}$, 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.

4.3 Lattice Attack on Secp256k1

We apply this lattice attack to the curve secp256k1 and assume that the Flush+Flush attack is perfect, which means we 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 is often solved through embedding itself into an SVP problem and using the polynomial time solution in practice. Therefore, we only demonstrate the result of using the SVP problem to solve the HNP instance.

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

We use the BKZ algorithm implemented in `fplll` [48] to solve the SVP problem. 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 about 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 [14].

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

5 RECOVER THE ECDSA PRIVATE KEY WITH EHNP

5.1 Extracting more Information

Assuming we get a perfect **Double-Add-Invert Chain**. Suppose in Double-Add-Invert Chain, the numbers of A is l, whose positions are separately $\lambda_i (1 \le i \le l)$. So we can easiy have

$$k = \sum_{i=0}^{l-1} k_i^{'} 2^{\lambda_i}$$

where

$$k_i^{'} \in \{-7, -5, -3, -1, 1, 3, 5, 7\}.$$

On the other hand, from the Invert chain, we can easily know that the $k_i^{'}$ is a positive integer or a negative integer. Suppose $k_i^{'} = (-1)^{h_i} k_i^*$, where $(-1)^{h_i}$ is the sign of $k_i^{'}$ which is known, $k_i^* \in \{1, 3, 5, 7\}$. Write $k_i^* = 2k_i + 1$, where $k_i \in \{0, 1, 2, 3\}$. Then we have

$$\begin{aligned} k &= \sum_{i=0}^{l-1} (-1)^{h_i} k_i^* 2^{\lambda_i} \\ &= \sum_{i=0}^{l-1} (-1)^{h_i} (2k_i + 1) 2^{\lambda_i} \\ &= \bar{k} + \sum_{i=0}^{l-1} (-1)^{h_i} k_i 2^{\lambda_i + 1} \end{aligned} \tag{8}$$

where $\bar{k} = \sum_{i=0}^{l-1} (-1)^{h_i} 2^{\lambda_i}$, h_i, λ_i are known and the only unknowns are $k_i \in \{0, 1, 2, 3\}$.

It is known that there are approximately $(\lceil logq \rceil + 1)/(\omega + 2) - 1 = 50.4$ non-zero digits in the Double-Add-Invert Chain when $\omega = 3$. From Eq.(8), there are approximately 50.4*2= 100.8 bits being unknown, which means the number of the known bits is about $257 - 100.8 = 156.2$ on average. In theory, we need at least two signatures to recover the 256-bit secrete key, as 2 is the least integer m such that $m \cdot 156.2 > 256$.

From above we can see, with our new Double-Add-Invert Chain, we can extract on average 156.2 bits of information per signature for the 256-bit elliptic curve which is about 1.5 times of the number of bits in [17] . In the next subsection, we will show that we can recover the secret key with only 2 signatures, which attains the optimal bound.

5.2 Find the target vector with new lattice

Similar to [17], we first translate the problem of recovering ECDSA secret key to the problem of EHNP. Then we further translate it to the problem of solving approximate SVP using lattice reduction algorithm.

On the other hand, in order to achieve the optimal bound, i.e., we only need 2 signatures and the corresponding Double-Add-Invert chain to recover the secret key. Some new ideas and techniques are introduced, which aims to reduce the attacking time or increase the success probability.

First and most importantly, we observe the components of target vector in [17] isn't balanced, so we reconstruct the lattice in [17], which reduces the length of target vector much, especially when the number of possible values of unknown k_i is small.

Then, we reduce the dimension of lattice and improve the success probability by guessing k_i. Finally, we observe that lattice reduction algorithm will recover different samples with different δ's, therefore, we provide some plausible reasonable δ's and use them to improve success probability.

In this section, we applies our attack to the secp256K1 curve. Three signatures are enough to recover the secret key with a high success probability no less than 73%. If we have only two signatures, we can recover the secret key with a success probability no less than 24.3%.

5.2.1 Reduction to EHNP Problem. Assuming we have a signature pair (r, s) of message m. Then we have the equation

$$\alpha r - sk + H(m) = 0 \mod q. \tag{9}$$

Substitue (9) with Equation (8), we can have that there exists an $h \in \mathbb{Z}$ such that

$$\alpha r - \sum_{i=1}^{l} ((-1)^{h_i} 2^{\lambda_i + 1} s) k_i - (s\bar{k} - H(m)) + hq = 0. \tag{10}$$

where $\alpha, k_i (1 \le i \le l)$ and h is unknown.

Assuming we got u signature pairs (r_i, s_i) of different messages $m_i (1 \le i \le u)$ with the same secret key α. Suppose we get the Double-ADD-Invert Chain of each ephemeral key.

From (11) we can easily have

$$\begin{cases} \alpha r_1 - \sum_{j=0}^{l_1-1}((-1)^{h_{1,j}}2^{\lambda_{1,j}+1}s_1)k_{1,j} - (s_1\bar{k}_1 - H(m_1)) + h_1^{'}q = 0 \\ \qquad\qquad \vdots \\ \alpha r_i - \sum_{j=0}^{l_i-1}((-1)^{h_{i,j}}2^{\lambda_{i,j}+1}s_i)k_{i,j} - (s_i\bar{k}_i - H(m_i)) + h_i^{'}q = 0 \\ \qquad\qquad \vdots \\ \alpha r_u - \sum_{j=0}^{l_u-1}((-1)^{h_{u,j}}2^{\lambda_{u,j}+1}s_u)k_{u,j} - (s_u\bar{k}_u - H(m_u)) + h_u^{'}q = 0 \end{cases} \tag{11}$$

where l_i is the number of non-zero digits of the i-th ephemeral key and $(-1)^{h_{i,j}}k_{i,j}$ is the j-th non-zero digit in the i-th signature, $\lambda_{i,j}$ is its position, $\bar{k}_i = \sum_{j=0}^{l_i-1}(-1)^{h_{i,j}}2^{\lambda_{i,j}} \mod q$ and $\alpha, h_i^{'}, k_{i,j}$ are unknown elements in the equations.

Given Equation (11), find $0 < \alpha < q$ and $0 \le k_{i,j} \le 2^{w-1} - 1$. We denote this problem DSA-EHNP.

5.2.2 Balancing the target vector. Next we will translate the DSA-EHNP problem to the problem of finding the short vector of a related lattice. Then we can find the secret key from the short vector. Comparing to the constructed lattice [17], the determinant of our new lattice is larger, while the target vector is shorter and more balanced, which makes the lattice reduction algorithm easier to find the target vector.

Notice that we have $k_{i,j} \in \{0, 1, \cdots, 2^{w-1}-1\}$. For $1 \le i \le u$ and $1 \le j \le l_i$, denote $\gamma_{i,j} = (-1)^{h_{1,j}}2^{\lambda_{1,j}+1}s_1r_i \mod q$, $c_{i,j} = (-1)^{h_{i,j}+1}2^{\lambda_{i,j}+1}s_ir_1 \mod q$, $\beta_i = r_1(H(m_i) - s_i\bar{k}_i) - r_i(H(m_1) - s_1\bar{k}_1) \mod q$. We can construct a lattice L spanned by the lines of the following matrix B in Equation (12) which is obtained by eliminating α in Equation (11). The parameter δ which will be discussed in section 5.2.3 is a proper value.

$$\mathbf{B} = \begin{pmatrix} q & & & & & & & & & & & & & \\ & \ddots & & & & & & & & & & & & \\ & & q & & & & & & & & & & & \\ \gamma_{2,1} & \cdots & \gamma_{\mu-1,1} & \frac{\delta}{3} & & & & & & & & & & \\ \vdots & & & & \ddots & & & & & & & & & \\ \gamma_{2,l_1} & \cdots & \gamma_{\mu-1,l_1} & & & \frac{\delta}{3} & & & & & & & & \\ c_{2,1} & & & & & & \frac{\delta}{3} & & & & & & & \\ \vdots & & & & & & & \ddots & & & & & & \\ c_{2,l_2} & & & & & & & & \frac{\delta}{3} & & & & & \\ & \ddots & & & & & & & & \ddots & & & & \\ & & c_{\mu,1} & & & & & & & & \frac{\delta}{3} & & & \\ & & \vdots & & & & & & & & & \ddots & & \\ & & c_{\mu,l_\mu} & & & & & & & & & & \frac{\delta}{3} & \\ \beta_2 & \cdots & \beta_u & \frac{\delta}{2} & \cdots & \frac{\delta}{2} & \frac{\delta}{2} & \cdots & \frac{\delta}{2} & \cdots & \frac{\delta}{2} & \cdots & \frac{\delta}{2} & \frac{\delta}{2} \end{pmatrix} \tag{12}$$

Suppose $h_i = r_1h_i^{'} - r_ih_1^{'}$, it is easy to check there exists

$$\begin{aligned} \mathbf{w} &= (h_2, \cdots, h_\mu, k_{1,1}, \cdots, k_{1,l_1}, \cdots, k_{u,1}, \cdots, k_{u,l_u}, 1)\mathbf{B} \\ &= (0, \cdots, 0, \tfrac{k_{1,1}}{3}\delta - \tfrac{\delta}{2}, \cdots, \tfrac{k_{1,l_1}}{3}\delta - \tfrac{\delta}{2}, \cdots, \\ &\quad \tfrac{k_{u,1}}{3}\delta - \tfrac{\delta}{2}, \cdots, \tfrac{k_{u,l_u}}{3}\delta - \tfrac{\delta}{2}, \tfrac{\delta}{2}) \in L(\mathbf{B}), \end{aligned}$$

and the Euclid norm of the vector $\mathbf{w}$ satisfies $\|\mathbf{w}\| \le \frac{\delta}{2}\sqrt{n-u+1}$, where n is the dimension of the lattice, i.e., $n = \sum_{i=1}^{u} l_i + u$.

The determinant of lattice $L(\mathbf{B})$ is $\|L\| = \frac{1}{2}q^{u-1}\cdot\delta^{n-u+1}(\frac{1}{3})^{n-u}$. The target vector $\mathbf{w}$ may not be the shortest vector, however, if we choose a appropriate value of δ, the target vector $\mathbf{w}$ which will be a pretty short vector which can be found by lattice reduction algorithm([30], [45], [14], [5]), thus, the secret key α can be recovered.

Comparing with the constructed lattice in [17]. Generally, the smaller $\frac{\|\mathbf{w}\|}{(|det(L)|)^{1/n}}$ is, the easier the lattice reduction algorithm is to find the target vector. Next, we will compare the value of $\frac{\|\mathbf{w}\|}{(det(L))^{\frac{1}{n}}}$ in the new lattice and [17]. The constructed lattice is similar to the lattice in [17] with two refinements. Firstly, we notice that if the diagonal entries of the matrix from line μ to line $n-1$ are $\frac{\delta}{7}$ instead of $\frac{\delta}{8}$ in [17], which not only make the determinant more larger , but also make the target vector more balanced and shorter. On the other word, $-\frac{\delta}{2} \le \frac{k_{i,j}}{8}\delta - \frac{\delta}{2} \le \frac{3\delta}{8}$ if $0 \le k_{i,j} \le 7$, while if we substitute $\frac{k_{i,j}}{8}$ by $\frac{k_{i,j}}{7}$, we have $-\frac{\delta}{2} \le \frac{k_{i,j}}{7}\delta - \frac{\delta}{2} \le \frac{\delta}{2}(0 \le k_{i,j} \le 7)$. In that case, the determinant of the new lattice is $(\frac{8}{7})^{n-u}$ times as large as the old lattice, while the expected length of the target vector will increase to $\sqrt{\frac{96}{77}}$ times, therefore, $\frac{\|\mathbf{w}\|}{(|det(L)|)^{1/n}}$ will decrease. Secondly, since we can get more information of the ephemeral key, i.e, for the lattice in [17], we have $0 \le d_{i,j} \le 7$, while now we have $0 \le d_{i,j} \le 3$. We can further change the diagonal entries of the matrix from line μ to line $n-1$ from $\frac{\delta}{7}$ to $\frac{\delta}{3}$, which increase the absolute value of the determinant of the lattice very much, however, slightly increase the length of the target vector. Therefore, we can find the target vector more easily when we use the newly constructed lattice.

The following experiments prove the validity of balancing. Firstly, we do tests using the way in [17] and the above refinements with 3 signatures, we can see the success probability is 73%, which is very high. Therefore, we try to recover the secret key with 2 signature in the following section. Our experiments in line 3 and line 4 and the following sections are based on the skills in [17] without merging which will drop the probability in experiment with 2 signatures. The results in Table 2 shows that balancing can effectively improve the success probability.

5.2.3 Guessing some $k_{i,j}$'s. In this section, we will analyse the way of guessing which can balance the success probability and time complexity of BKZ. Considering that the set of $k_{i,j}$ is small, we guess the value of $k_{i,j}$ to improve the success probability. Each $k_{i,j}$ has 2 bits information, thus, the expectation of the number of guessing to gain the right $k_{i,j}$ is $\frac{1}{4}(1+2+3+4) = \frac{5}{2}$. Every time we guess $k_{i,j}$, the expected

time complexity will increase to $\frac{5}{2}$ times and the dimension will decrease 1.

Guessing will increase the time complexity, however, it can also improve success probability. From experiment results based on the skill of balancing in Table 3, we can observe that the time complexity will increase to 3.7/34.2 times when we guess 2/4 $k_{i,j}$'s, while the success probability will expand 2.7/4.4 times respectively. Moreover, comparing to the way of not using guessing with larger block size(35), the skill of guessing with block size(30) will be more efficient. Therefore, in the next section 5.2.4, we will further discuss how to improve the success probability based on the technology of balancing and guessing.

5.2.4 Choosing different δ's. Finally, based on the experimental phenomena: 1. the success probability will change with different δ's, 2. different samples will be recovered if we select different δ's, we analyse how to select the parameter δ and utilize different δ's to improve the success probability.

We observe some situations that lattice can produce vector set T consisting of shorter vectors than target vector. Based on these observations, we expect to reduce the size of T by selecting suitable parameter δ to improve success probability. The followings are some heuristic δ's.

1. If the target vector is enough short, we can recover it with high success probability. Therefore, we choose $\delta_0 = \frac{2}{\sqrt{n-\mu+1}}$ to guarantee any lattice vector $\mathbf{a}$ which exists $\mathbf{a}_i \neq 0$, for $i \in [0, m-1]$, is much longer than target vector $\mathbf{w}$. Because, for $i \in [0, m-1]$, when $\mathbf{a}_i \neq 0$, $|\mathbf{a}_i| \geq 1$ then $\|\mathbf{a}\| \geq 1 \geq \|\mathbf{w}\|$. Therefore, by setting $\delta = \frac{2}{\sqrt{n-\mu+1}}$, the size of set T will decrease, which will improve the probability of lattice reduction algorithm.

2. We try some other δ's, just like $2\delta_0$, $\frac{\delta_0}{2}$ and so on because we notice that different δ's may recover different samples. For example, for 5 examples, setting $\delta = \delta_0$ will recover the secret key of 1-th, 3-th and 4-th samples, however, the samples recovered will be 1, 2 if we let $\delta = \frac{\delta_0}{2}$. Therefore, for the same samples, if we choose several δ's to construct different lattices, we can improve the success probability.

Of course, this method will enlarge the time complexity, however, comparing with the improvement of success probability, the increment of time complexity is acceptable. The related data basing on the ways of balancing and guessing and different δ's can be found in Table 4. We can see the probability will increase to 24.3% when using 3 different δ's, however, the time complexity will approximately increase by 2 times. Compared with the experiment using guessing 4 $k_{i,j}$'s in Table 3, the way of combining guessing 2 $k_{i,j}$'s and different δ's is more efficient.

μ	samples	balance	block	probability	time(s)
3	1000	Yes	30	73%	34.7
2	1000	No	30	0%	126.7
2	1000	Yes	30	5.7%	154.0

Table 2: The advantage of balancing

"samples" represents the number of samples, "block" is the blocksize of BKZ and "time(s)" is the time of BKZ on each sample.

μ	samples	guessing	block	δ	probability	time(s)
2	1000	2	30	δ_0	15.3%	566.7
2	1000	4	30	δ_0	25.3%	5260.4
2	1000	0	35	δ_0	9%	3387

Table 3: Balancing and guessing

"guessing" represents the number of $k_{i,j}$ guessed, "δ_0" will be explained in section 5.2.4.

μ	samples	guessing	block	δ	probability	time(s)
2	1000	2	30	δ_0	15.3%	566.7
2	1000	2	30	$\frac{\delta_0}{2}$	14.6%	550.0
2	1000	2	30	$2\delta_0$	13%	516.7
total					24.3%	1633.4

Table 4: Balancing, guessing and different δ's

The last line records the total success probability and the average time using three different δ's .

6 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 obtain the Double-ADD or Double-Add-Invert chain and extract partial information about k. Then, the private key recovery from incomplete information of k is transformed into a problem that can be solved by lattice reduction, such as the HNP or EHNP problem. Finally, through being converted to the CVP/SVP problem in the lattice, the ECDSA private key is recovered.

Benger et al. [8] 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 constructed an HNP problem and successfully recovered the private key by solving the CVP/SVP problem converted from the HNP problem. The number of bits extracted is only an average of 2, 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. [50] improved Benger's attack and used the information from the consecutive non-zero digits whose positions are larger than the half of the length of the Double-Add chain extracted from the Flush+Reload attack. Then they constructed an HNP instance using this information and successfully recovered the private key. It is able to extract 47.6 bits per signature on average for the secp256k1 curve and recover the private key with 13 signatures.

Fan et al.[17] 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

Table 5: Comparison with previous attack methods

Methods	Exploited information	HNP or EHNP	# of bits	# of signatures
Benger et al. [8]	LSB	HNP	2	200
Van de Pol et al. [50]	half positions of non-zero digits	HNP	47.6	13
Fan et al. [17]	all positions of non-zero digits	EHNP	105.8	4
Wang et al. [51]	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

105.8 bits per signature for the secp256k1 curve. With some optimization only 4 signatures are needed to recover the private key with the probability being 8%.

Wang et al. [51] 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.

We compare these attacks with ours in several aspects, and the result is shown in Table 5. 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 156.2 bits per signature for the secp256k1 curve. The number of signatures needed to recover the private key is only 2.

7 DISCUSSION

7.1 The Imperfect Data from the Cache Side Channel

Our lattice attack analysis is based on the assumption of the perfect Double-Add-Invert Chain, which means no errors exist in the chain. But in practical, achieving the perfect chain is almost impossible from the cache side channels. Van de Pol et al. [50] analysed the actual cache side channel data obtained from the Flush+Reload attack. Their statistical results show that 57.7% of all the obtained Double-Add Chains are perfect. We also can monitor the functions using the same methods within the Flush+Reload attack, so the probability of the perfect Double-Add-Invert Chain is also 57.7%, the same as Van de Pol's. Thus, the actual number of signatures needed to recover the 256-bit ECDSA private key is 4.

While in our experiments, we use the Flush+Flush attack to obtain the Double-Add-Invert Chain. Compared with Flush+Reload, the Flush+Flush is more easily to have errors, because the distributions of the execution time of the `clflush` instruction on cached and non-cached memory have some overlapping and the difference in timing of this instruction is less. So the probability to obtain the perfect chain is less than 57.7%. We can re-measure the probability of obtaining the perfect chain using the same method with Van de Pol and achieve the actual number of signatures needed.

7.2 The Precision Of The Cache Side Channel Attacks

aaaa

8 RELATED WORK

8.1 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 [10] 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 [24] extended the work of Boneh and Venkatesan [10] 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 [37] 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 [38]. Liu and Nguyen [32] improved the results that only 2 LSBs were required for breaking a 160-bit DSA key. In 2014, Benger et al. [8] extended the technique in [37] 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. [50] 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. [17] 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 [12] 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 [24] with collecting $2,600$ signatures ($8K$ with noise). Analogously, Acıiçmez et al. [3] 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 [13] 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. [15] 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. [8] and Van de Pol et al. [50] used the Flush+Reload technique to acquire the information of the ephemeral key. Allan et al. [6] improved the results in [50] 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. [18] 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. [42] 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. [54] extended the attack in [38] to SM2 Digital Signature Algorithm (SM2-DSA), which is a Chinese version of ECDSA.

8.2 Cache Side Channel Attacks

In 2002, Page [40] 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. [49] 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 [9], and it statically analyzed the relation between the overall execution time and lookup table indexes affected by the cache behavior. Bonneau and Mironov [11] showed how to exploit cache collisions when indexing the AES lookup tables to recover the AES secret key. In 2006, Osvik et al. [39] 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 [4] 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. [22] 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 [26] exploit the Flush+Reload technique to attack the AES.

Percival [41] presented the first cache attack on RSA using the data cache collision. Acıiçmez in [2] first exploited that the instruction cache leaked information when performing RSA encryption. Brumley and Hakala [12] 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. [3] 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 [53]. Then, again Yarom et al. used the Flush+Reload technique to recover the secret key from an ECDSA signature algorithm [52].

Cloud computing systems also can be attacked using the cache side channel attacks. In 2009, Ristenpart et al. [44] presented the methods to co-locate an attacker's virtual machine (VM) with a potential victim's 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 victim's 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 [26] and RSA[53] in the cross-VM scenario. Also, the Prime+Probe method was also adapted to work in virtualized environments by Zhang et al. [55] and Liu [31] to recover ElGamal encryption keys. Irazoqui et al. [25] recovered AES keys in virtualized environments with Bernstein's attack. Benger et al. [8] and Van de Pol et al. [50] demonstrated the viability of the Flush+Reload technique to recover ECDSA encryption keys. Fan et al. [17] proposed a new way of extracting and utilizing information from the Flush+Reload attack. Flush+Flush [21] and Prime+Abort [16] 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.

9 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.

ACKNOWLEDGMENTS

To Robert, for the bagels and explaining CMYK and color spaces.

REFERENCES

[1] [n.d.]. OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[2] Onur Acıiçmez. 2007. Yet another MicroArchitectural Attack: Exploiting I-Cache. In *Proceedings of the 2007 ACM workshop on Computer Security Architecture, CSAW*. ACM, 11–18.

[3] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New Results on Instruction Cache Attacks. In *Cryptographic Hardware and Embedded Systems, CHES 2010*. Springer Berlin Heidelberg, 110–124.

[4] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-Driven Cache Attacks on AES (Short Paper). In *Information and Communications Security*. Springer Berlin Heidelberg, 112–121.

[5] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. 2019. The General Sieve Kernel and New Records in Lattice Reduction. In *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part II*. 717–746.

[6] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. 2016. Amplifying Side Channels Through Performance Degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16)*. ACM, 422–435.

[7] American National Standards Institute. 2005. *ANSI X9.62-2005, Public Key Cryptography for the Financial Services Industry : The Elliptic Curve Digital Signature Algorithm (ECDSA)*.

[8] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2014. "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*. Springer Berlin Heidelberg, 75–92.

[9] Daniel J Bernstein. 2005. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[10] Dan Boneh and Ramarathnam Venkatesan. 1996. Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *Advances in Cryptology — CRYPTO '96*. Springer Berlin Heidelberg, 129–142.

[11] Joseph Bonneau and Ilya Mironov. 2006. Cache-Collision Timing Attacks Against AES. In *Cryptographic Hardware and Embedded Systems - CHES 2006*. Springer Berlin Heidelberg, 201–215.

[12] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In *Advances in Cryptology - ASIACRYPT 2009 - 15th International Conference on the Theory and Application of Cryptology and Information Security*. Springer Berlin Heidelberg, 667–684.

[13] Billy Bob Brumley and Nicola Tuveri. 2011. Remote Timing Attacks Are Still Practical. In *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security*. Springer Berlin Heidelberg, 355–371.

[14] Yuanmi Chen and Phong Q. Nguyen. 2011. 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*. Springer Berlin Heidelberg, 1–20.

[15] Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. 2013. 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*. Springer Berlin Heidelberg, 435–452.

[16] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, 51–67.

[17] Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. 2016. Attacking OpenSSL Implementation of ECDSA with a Few Signatures. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 1505–1515.

[18] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 1626–1638.

[19] Daniel M. Gordon. 1998. A Survey of Fast Exponentiation Methods. *Journal of Algorithms* 27, 1 (1998), 129 – 146.

[20] Dahmun Goudarzi, Matthieu Rivain, and Damien Vergnaud. 2017. Lattice Attacks Against Elliptic-Curve Signatures with Blinded Scalar Multiplication. In *Selected Areas in Cryptography – SAC 2016*. Springer International Publishing, 120–139.

[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: A fast and stealthy cache attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 279–299.

[22] D. Gullasch, E. Bangerter, and S. Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *32nd IEEE Symposium on Security and Privacy, S&P 2011*. 490–505.

[23] Martin Hlavác and Tomás Rosa. 2006. Extended Hidden Number Problem and Its Cryptanalytic Applications. In *Selected Areas in Cryptography (SAC)*. 114–133.

[24] N. A. Howgrave-Graham and N. P. Smart. 2001. Lattice Attacks on Digital Signature Schemes. *Designs, Codes and Cryptography* 23, 3 (Aug 2001), 283–290.

[25] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. 2014. Fine Grain Cross-VM Attacks on Xen and VMware. In *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*. 737–744.

[26] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. 2014. Wait a Minute! A fast, Cross-VM Attack on AES. In *Research in Attacks, Intrusions and Defenses*. Springer International Publishing, 299–319.

[27] J. Callas, L. Donnerhacke, H. Finney, D. Shaw and R. Thayer. 2007. *OpenPGP Message Format (RFC 4880)*.

[28] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security* 1, 1 (Aug 2001), 36–63.

[29] Kenji Koyama and Yukio Tsuruoka. 1992. Speeding up Elliptic Cryptosystems by Using a Signed Binary Window Method. In *Advances in Cryptology — CRYPTO' 92*. Springer Berlin Heidelberg, 345–357.

[30] A. K. Lenstra, H. W. Lenstra, and L. Lovász. 1982. Factoring polynomials with rational coefficients. *Math. Ann.* 261, 4 (Dec 1982), 515–534.

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. 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*. IEEE Computer Society, 605–622.

[32] Mingjie Liu and Phong Q. Nguyen. 2013. Solving BDD by Enumeration: An Update. In *Topics in Cryptology - CT-RSA 2013 - The Cryptographers' Track at the RSA Conference 2013*. Springer Berlin Heidelberg, 293–309.

[33] Atsuko Miyaji, Takatoshi Ono, and Henri Cohen. 1997. Efficient Elliptic Curve Exponentiation. In *Proceedings of the First International Conference on Information and Communication*

Security (ICICS '97). Springer-Verlag, 282–291.
[34] Satoshi Nakamoto. 2008. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf
[35] National Institute of Standards and Technology. 2013. *FIPS PUB 186-4 Digital Signature Standard (DSS)*.
[36] Phong Q. Nguyen. 2001. The Dark Side of the Hidden Number Problem: Lattice Attacks on DSA. In *Cryptography and Computational Number Theory*. Birkhäuser Basel, 321–330.
[37] Phong Q. Nguyen and Igor E. Shparlinski. 2002. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *Journal of Cryptology* 15, 3 (Jun 2002), 151–176.
[38] Phong Q. Nguyen and Igor E. Shparlinski. 2003. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Designs, Codes and Cryptography* 30, 2 (Sep 2003), 201–217.
[39] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology - CT-RSA 2006 - The Cryptographers' Track at the RSA Conference 2006*. Springer Berlin Heidelberg, 1–20.
[40] Dan Page. 2002. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptology ePrint Archive* 2002 (2002), 169.
[41] Colin Percival. 2005. CACHE MISSING FOR FUN AND PROFIT.
[42] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. Make Sure DSA Signing Exponentiations Really Are Constant-Time. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, 1639–1650.
[43] W Rankl. 2007. Smart Card Applications: Design Models for Using and Programming Smart Cards. (01 2007).
[44] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. 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 (CCS '09)*. ACM, 199–212.
[45] C. P. Schnorr and M. Euchner. 1994. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming* 66, 1 (Aug 1994), 181–199.
[46] Jerome A. Solinas. 2000. Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography* 19, 2 (Mar 2000), 195–249.
[47] T. Dierks and E. Rescorla. 2008. *The Transport Layer Security (TLS) Protocol Version 1.2 (RFC 5246)*.
[48] The FPLLL development team. 2016. fplll, a lattice reduction library. (2016). https://github.com/fplll/fplll Available at https://github.com/fplll/fplll.
[49] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES Implemented on Computers with Cache. In *Cryptographic Hardware and Embedded Systems - CHES 2003*. Springer Berlin Heidelberg, 62–76.
[50] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. 2015. Just a Little Bit More. In *Topics in Cryptology - CT-RSA 2015 - The Cryptographers' Track at the RSA Conference 2015*. Springer International Publishing, 3–21.
[51] Wenbo Wang and Shuqin Fan. 2017. Attacking OpenSSL ECDSA with a small amount of side-channel information. *Science China Information Sciences* 61, 3 (Aug 2017), 032105.
[52] Yuval Yarom and Naomi Benger. 2014. Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack. *IACR Cryptology ePrint Archive* 2014 (2014), 140.
[53] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, 719–732.
[54] K. Zhang, S. Xu, D. Gu, H. Gu, J. Liu, Z. Guo, R. Liu, L. Liu, and X. Hu. 2017. Practical Partial-Nonce-Exposure Attack on ECC Algorithm. In *2017 13th International Conference on Computational Intelligence and Security (CIS)*. 248–252.
[55] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. 2012. 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*. ACM, 305–316.