IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 4, pp. 490–526. DOI:10.46586/tches.v2022.i4.490-526

One Truth Prevails: A Deep-learning Based Single-Trace Power Analysis on RSA–CRT with Windowed Exponentiation

Kotaro Saito, Akira Ito, Rei Ueno and Naofumi Homma

Tohoku University, 2–1–1 Katahira, Aoba-ku, Sendai-shi, Miyagi, 980-8577, Japan,
rei.ueno.a8@tohoku.ac.jp, naofumi.homma.c8@tohoku.ac.jp

Abstract. In this paper, a deep-learning based power/EM analysis attack on the state-of-the-art RSA–CRT software implementation is proposed. Our method is applied to a side-channel-aware implementation with the Gnu Multi-Precision (MP) Library, which is a typical open-source software library. Gnu MP employs a fixed-window exponentiation, which is the fastest in a constant time, and loads the entire precomputation table once to avoid side-channel leaks from multiplicands. To conduct an accurate estimation of secret exponents, our method focuses on the process of loading the entire precomputation table, which we call a dummy load scheme. It is particularly noteworthy that the dummy load scheme is implemented as a countermeasure against a simple power/EM analysis (SPA/SEMA). This type of vulnerability from a dummy load scheme also exists in other cryptographic libraries. We also propose a partial key exposure attack suitable for the distribution of errors in the secret exponents recovered from the *windowed* exponentiation. We experimentally show that the proposed method consisting of the above power/EM analysis attack, as well as a partial key exposure attack, can be used to fully recover the secret key of the RSA–CRT from the side-channel information of a single decryption or a signature process.

Keywords: Side-channel attack · Deep learning · RSA–CRT · Partial key exposure attack · Gnu MP · OpenSSL · Botan · Libgcrypt

1 Introduction

1.1 Background

Side-channel attack on RSA–CRT. Modular exponentiation plays an essential role in a public key cryptosystem (PKC) such as RSA, digital signature algorithm (DSA), and elliptic curve cryptosystems (ECCs). In RSA decryption/signing and the DSA, the exponent is usually a secret key and a nonce, respectively. Hence, many side-channel attacks on RSA aim at directly recovering the exponent from power or electromagnetic (EM) traces. Among them, single-trace attacks (also known as horizontal attacks) have a significant advantage over multiple-trace attacks in the context of PKC. This is because PKC is not frequently executed unlike symmetric primitives, as in hybrid cryptosystems using key and data encapsulation mechanisms. In fact, many previous studies have developed single-trace attacks on RSA and ECCs (*e.g.*, [CFG+10, BJPW13, KDKL17]). One major challenge of single-trace attacks is to estimate the secret exponent with a high accuracy. A secret exponent estimated through a power/EM trace usually contains errors from noise included in the trace. However, in attacking an RSA–CRT, a partial key exposure attack (represented by Heninger–Shacham algorithm [HS09]) is frequently used to recover the

Received: 2022-04-15 Accepted: 2022-06-15 Published: 2022-08-31

correct secret exponent from its noisy version. Thus, studies on a single-trace side-channel attack on RSA–CRT include two major topics: (1) estimating the secret exponent(s) from a side-channel trace as correctly as possible through a side-channel attack, and (2) removing the errors in estimated exponents as correctly and robustly as possible through a partial key exposure attack.

Deep-learning based side-channel attack (DL-SCA). A deep-learning based power/EM analysis attack, the so-called Deep-Learning based Side-Channel Analysis (DL-SCA) has received significant attention. Since the pioneering study on DL-SCA was conducted [MPP16], many DL-SCA related studies have mainly focused on block ciphers, and have extensively revealed the significant advantage of DL techniques in side-channel attacks, even on implementations with masking and/or random delay countermeasures [BPS+20, AG01, DBN+01, CK09]. In addition, some DL-SCAs have been developed for RSA, ECC, and post-quantum cryptography (PQC) [CCC+19, LLQ+20, PCBP20, UXT+22]. In particular, DL-SCAs on RSA employ DL techniques for correctly estimating the secret exponent. In [CCC+19, ZBHV21], Carbone *et al.* and Zaid *et al.* showed a DL-SCA on a non-CRT RSA with several side-channel countermeasures, namely, square-multiply always exponentiation, and message-, exponent-, and remainder-blinding, under the condition of an authentication co-processor. In [LLQ+20], Lei *et al.* also showed a DL-SCA on RSA–CRT implementation with left-to-right exponentiation, which employs jittering and power consumption balancing as countermeasures. In addition, in [PCBP20], Perin *et al.* showed a non-profiling DL-SCA on ECC with a Montgomery ladder, conditional swap (cswap), and coordinate re-randomization, which is likely applied to the RSA. These existing studies demonstrated the capability of a DL-SCA on *binary* modular exponentiation (*i.e.*, left-to-right, square-multiply always, and a Montgomery ladder) with several side-channel countermeasures. Also note that the existing studies on DL-SCAs have mainly focused on estimating the secret exponent(s) more correctly, but have not evaluated the entire attack flow of the full key (*i.e.*, exponent) recovery from a noisy estimated version.

Open-source RSA implementations. The applicability of DL-SCAs remains unclear, particularly on the practical RSA–CRT implementations included in open-source libraries. In practice, the Chinese remainder theorem (CRT) is usually employed in RSA decryption and signing for reducing the computation time. In addition, some modern open-source cryptographic library implementations employ a *windowed* exponentiation (*i.e.*, fixed- or sliding-window exponentiation) owing to its better performance than binary exponentiation. In fact, the sliding-window is known to be the fastest exponentiation algorithm for implementing the RSA, and the fixed-window is the fastest constant-time algorithm [Koç95]. Moreover, open-source cryptographic libraries are commonly equipped with countermeasure(s) against side-channel attacks, such as exponent blinding and a dummy load, for hiding the temporal window values (*i.e.*, multiplicand addresses). This is mainly because such libraries (particularly those providing TLS) should be robust to cache side-channel attacks using Flush+Reload and Prime+Probe [YF14, KHF+19, YGH17, BBG+17]. This indicates that these open-source library implementations would be (somehow) resistant to a simple power analysis (SPA)-type attacks, as the above countermeasures and windowed exponentiations are known to be effective for an SPA prevention. However, it is unclear how effective DL-SCAs are for these implementations. To the best of our knowledge, only a few DL-SCAs for practical implementations were evaluated in [WPB19, WCPB20], which targeted a 4-bit fixed-window EdDSA implementation in WolfSSL, and there is no report on RSA implementations with windowed exponentiation. Thus, the evaluation of DL-SCAs on such practical implementations included in major open-source libraries are important from both academic and practical perspectives.

RSA–CRT partial key exposure attack. A partial key exposure attack for random bit leaks was first presented by Heninger and Shacham [HS09] and has been widely employed in many side-channel attacks including cold boot attacks and cache attacks [HSH+08,BBG+17]. The Heninger–Shacham algorithm recovers the RSA–CRT secret primes and exponents from partial information of the secret exponents (and optionally primes)[1] using a branch-and-prune method, which discards candidates of unknown bits in order of the least to most significant bits according to RSA–CRT constraint equations. In [HMM10], Henecka *et al.* extended the branch-and-prune attack to the bit flip leak model, where the attacker knows all bits of the secret exponent (and optionally primes) with random bits flipped with an error probability. Whereas the Heninger–Shacham algorithm is used for cache attacks on RSA–CRT with sliding-window exponentiation [BBG+17], as an example, Henecka's algorithm is useful for an SPA on the RSA–CRT. One major limitation of Henecka's attack is that bit flips (*i.e.*, errors of partial information) are assumed to be uniformly distributed. If flips are continuous over several bits, their algorithm often erroneously discards the correct path, which reduces the success rate of attack. Intuitively, Henecka's leak model fits the SPA leakage of a binary exponentiation but does not always represent the leakage of a windowed exponentiation, where window width-wise bit flips may occur. A more appropriate model should be considered in attacking a windowed exponentiation.

1.2 Our contributions

In this paper, the first *single-trace* power analysis attack on RSA–CRT using *windowed exponentiation*, which is applied in several open-source libraries, is presented with the help of the DL technique. The contribution of this paper is fourfold.

1. We present a new DL-SCA that can be applied to some modern open-source RSA–CRT implementations employing a windowed exponentiation and dummy load scheme for counteracting conventional side-channel attacks. Such a dummy load scheme is employed for example to counter attacks using the load address [MD99, IIT02, KDKL17] and Prime+Probe. The main idea of the proposed method is distinguishing the true and dummy loads in selecting the operand using the DL technique. Many windowed exponentiation implementations load all operands and then discard incorrect operations. In other words, the proposed attack reduces the temporal window value estimation to a 2-classification of true/dummy load, which yields a significant improvement of estimation accuracy compared to a straightforward 2^w-classification, and a reduction of costs to develop and train an NN model. Our methodology would be applicable to many algorithms/implementations with dummy loads, as our methodology focuses on only dummy/true load and does not exploit algorithm/implementation-specific features at all.

2. We propose a new partial key exposure attack algorithm for RSA–CRT with a windowed exponentiation. The proposed algorithm considers the characteristics of estimation errors in the above DL-SCA on a windowed exponentiation. The proposed algorithm employs a double-ended queue with a heuristic-based priority, and in contrast to the existing algorithms, can recover the full secret exponent under the condition that a noisy version is estimated through the above DL-SCA. As a result, the proposed attack consisting of the above DL-SCA and partial key exposure attack algorithm makes it possible to fully recover the secret primes and exponents (*i.e.*, secret key) from only one power/EM trace.

3. We experimentally demonstrate and validate that the proposed attack can recover the secret key of 1,024- and 2,048-bit RSA–CRT implemented using the open-source Gnu

[1] In the Heninger–Shacham attack, the attacker knows the random bits of secret exponents and primes, and has no information about remaining bits.

Multi-Precision (MP) Library, which is a major open-source library of arithmetic operations and has been widely adopted. Moreover, GnuMP may be used for building major open-source TLS implementations such as OpenSSL[2] and GnuTLS (and its requirement library Nettle). In addition, its practicality is outstanding because of its high performance and because it implements some countermeasures against SCAs (including cache and SPA-like attacks). Thus, the vulnerability analysis of Gnu MP is essential for both academic and practical studies. In addition, analyzing other open-source libraries, we also show that OpenSSL[3], Botan, and Libgcrypt implementations (partially) have the same vulnerability. The source codes used in our experiment will be publicly available for the third-party reproducibility.

4. We present the *randomized dummy load* as a possible countermeasure against the proposed attack. The countermeasure can surely make the proposed single-trace attack difficult, and cooperate with the conventional ones, such as the exponent blinding, to prevent multiple-trace attacks (*e.g.*, [HMA+10]).

Responsible disclosure. Although open-source cryptographic software libraries do not necessarily consider embedded device implementation or power/EM attacks, we have disclosed our findings to the related library developers.

Repository for our source code. The source codes used in the experiment for our attack are publicly available at https://github.com/ECSIS-lab/one_truth_prevails.

1.3 Paper organization

The rest of this paper is organized as follows. Section 2 describes RSA–CRT implementation in open-source libraries. In particular, we will focus on the exponentiation implementation of Gnu MP Library. Besides, we briefly describe the existing DL-SCA and partial key exposure attack algorithms on RSA (and ECC). Section 3 presents our proposed attack on RSA–CRT implementation with windowed exponentiation and dummy load based countermeasure. Section 4 demonstrates the validity of our method through experimental attacks. Section 5 discusses the impact of our method on other open-source libraries and the reason why the proposed DL-SCA successfully works. We also provide a countermeasure against the proposed DL-SCA. Section 6 concludes the paper.

2 Preliminaries

2.1 RSA–CRT

Let (e, N) be the public key for the RSA cryptosystem, where e is public exponent usually given by $2^{16}+1$, and N is a public modulus given as a product of distinct primes p and q (*i.e.*, $N = pq$). Let (d, p, q) be a secret key, where d is a secret exponent and (p, q) are secret primes. An RSA encryption/decryption is computed using modular exponentiation. Textbook RSA encryption and decryption are expressed as

$$c = m^e \bmod N,$$
$$m = c^d \bmod N,$$

respectively, where m is a plaintext, and c is the corresponding ciphertext. In addition, the RSA signing is expressed as

$$\sigma = H^d \bmod N,$$

[2]OpenSSL provides an option to adopt Gnu MP for its backend.
[3]This denotes the stand-alone OpenSSL without Gnu MP codes.

where H is a fingerprinting of a signing entity, and σ is the corresponding signature. The verification of σ examines whether it holds, *i.e.*,

$$H = \sigma^e \bmod N.$$

In this paper, we use a notation for RSA decryption.

RSA implementation usually employs the CRT for computational efficiency. The RSA decryption with CRT is called RSA–CRT. The RSA–CRT implementation preserves $(p-1, q-1, q_p^{-1}, d_p, d_q)$ as the secret key in addition to (p, q, d), where $q_p^{-1} = q^{-1} \bmod p$ $d_p = d \bmod p-1$ and $d_q = d \bmod q-1$, and decrypts a ciphertext c as follows:

$$\begin{aligned} c_p &= c \bmod p-1, \\ c_q &= c \bmod q-1, \\ m_p &= {c_p}^{d_p} \bmod p, \\ m_q &= {c_q}^{d_q} \bmod q, \\ m &= (({m_p} - {m_q}){q_p}^{-1} \bmod p) \cdot q + m_q. \end{aligned}$$

In CRT–RSA decryption, the bit length of the key and operand become half of the textbook RSA, which allows for a reduction of the computational complexity.

2.2 Windowed exponentiation in open-source implementations

Fixed-window. Fixed-window (also known as 2^w-ary) exponentiation is the fastest constant-time modular exponentiation algorithm. Algorithm 1 is a left-to-right fixed-window modular exponentiation with a base c, an l-bit exponent d, a modulus N, and a window size w. In lines 1–4, multiplicands c^0, c^1, ..., and c^{2^w-1}, are precomputed. Lines 7–15 constitute the main loop, and in each of these lines, we apply w squaring followed by a multiplication. More precisely, line 9 determines the multiplicand according to the temporal window; lines 10–12 apply w squaring; and line 13 conducts a multiplication with a multiplicand corresponding to the temporal window value. In the fixed-window algorithm, the squaring–multiplication operational sequence is not exponent-dependent, and therefore an attacker that observes the sequence (*e.g.*, SPA/SEMA attacker) cannot obtain any information regarding the secret exponent in principle. Owing to its constant-time feature and high performance, many open-source implementations employ a fixed-window exponentiation. For example, Gnu MP [noa22a, Gra] provides a fixed-window modular exponentiation with numerous computational optimizations for an arbitrary precision arithmetic and is occasionally adopted to build a cryptographic library such as OpenSSL [noa21b], GnuTLS [noa22b], and Nettle [noa22c].

Dummy load to hide temporal window value. The fixed-window exponentiation is resistant to SPA-like attacks, which rely on an observation of the squaring–multiplication sequence but cannot obtain information regarding multiplicands. In other words, windowed exponentiation is vulnerable if the attacker can know the multiplicand value at the multiplication (*i.e.*, temporal window value b). Some side-channel attacks conducted to estimate the temporal window value have been reported thus far. Alam *et al.* [AKD+18] showed an EM analysis attack that exploits the window value acquisition in a previous version of OpenSSL. Yarom *et al.* [YGH17] also showed a cache attack that estimates a partial value of the secret exponents by recovering a portion of the window value. These attacks showed that the temporal window value can be leaked through cache side-channel due to the difference of memory addresses storing the multiplicands. To prevent such leaks, some fixed-window exponentiation implementations, including Gnu MP, utilize dummy loading. For the multiplication, all multiplicands are loaded once, and then

Algorithm 1 Left-to-right fixed-window modular exponentiation

Input: Base c, exponent $d = (d_{l-1}d_{l-2}\dots d_0)_{(2)}$, modulus N, and window size w
Output: **int** $m = c^d \mod N$;

```
1:  c_0 ← 1
2:  for t from 1 to 2^w − 1 do
3:      int c_t ← c_{t−1} · c mod N;          ▷ Precomputation of multiply operands
4:  end for
5:  int m ← 1;
6:  int λ ← l − 1;
7:  while λ ≥ 0 do
8:      int μ ← max(λ − w + 1, 0);
9:      int b ← (d_λ ... d_μ)_(2);            ▷ Acquire window value
10:     for z from 1 to λ − μ + 1 do
11:         m ← m · m mod N;                  ▷ Squaring
12:     end for
13:     m ← m · c_b mod N;                    ▷ Multiplication
14:     λ ← μ − 1;
15: end while
16: return m;
```

only the true multiplicand is preserved and other multiplicands are discarded as dummy loads. Because such a dummy load scheme performs a load operation for all operands, it will hide the secret temporal window value from power/EM attackers that attempt to observe the dependency of the instruction flow on the temporal window value from the implementation (*i.e.*, offer the resistance to cache attacks exploiting the data dependency, such as Prime+Probe [LYG+15, YGH17]).

Existing DL-SCA to distinguish true/dummy load. A true/dummy load is sometimes used as a countermeasure against SCAs, some attacks that detect a true/dummy load have been reported. Regarding DL-SCA, in [LH20a], Lee and Han demonstrated an attack on AES software implementation with dummy-load and shuffle based countermeasures. They designed a multi-label classification CNN to detect dummy Sbox operations followed by a power analysis, e.g., differential power analysis (DPA), and demonstrated that the CNN-based detection had an advantage in accuracy over the conventional method depicted in [LH20b] that adopted a bounded collision detection criterion (BCDC) [DLLM15]. On the other hand, their multi-label CNN was designed for attacking the specific AES implementation; therefore, the applicability to other ciphers (*i.e.*, RSA) and implementations was unclear. In addition, the following key recovery phase depending on the cipher and/or implementation was not known. Note that the contributions made in this study include estimating the secret exponents from detected dummy loads and a partial key exposure attack algorithm that achieves a full key recovery, in addition to an efficient DL-based dummy load detection.

Remark 1. In addition to a fixed-window exponentiation, some sliding-window implementations also utilize such a dummy load, probably to prevent the aforementioned leak of a temporal window value (*e.g.*, Libgcrypt [Pro21]). Although this paper presents the proposed attack with an application to a Gnu MP fixed-window implementation, our attack can be applied to some sliding-window implementations through a dummy load scheme, as discussed in Section 5.2.

Algorithm 2 Multiplicand load in Gnu MP modular exponentiation [Gra]

Input: Precomputed table $(c^0, c^1, \ldots c^{2^w-1})$, window size w, and window value b
Output: Multiplicand $s = c^b$
1: **for** i from 0 to $2^w - 1$ **do**
2: **int** mask $\leftarrow$ MakeBitMask(i, b); ▷ MakeBitMask(i, b) returns $(11\ldots1)_{(2)}$ if $i = b$, else 0
3: **int** $s \leftarrow (s\&\neg\text{mask}) \mid (c^i\&\text{mask})$;
4: **end for**
5: **return** s;

Table 1: Modular exponentiation in Gnu MP (when $m = c^{401} \mod N$ and window size $w = 3$)

Exponent (binary expression)	110010001		
Window value	110	010	001
Squaring (S)/Multiplication (M)	SSSM	SSSM	SSSM
Multiplicand	c^6	c^2	c^1
True (T)/Dummy (D) loading	DDDDDDTD	DDTDDDDD	DTDDDDDD

Gnu MP implementation of multiplicand load function. For example, Algorithm 2 shows the multiplicand load function in Gnu MP, and Table 1 shows an example computation of $m = c^{401} \mod N$. The For loop in lines 1–4 applies multiplicand loading in ascending index order; that is, the i-th loop loads c^i. Line 2 generates a bitmask, the value of which is set to 1^w when the loop count i equals the window value b (*i.e.*, when the loaded multiplicand is true); otherwise, it is set to 0^w. Line 3 then computes the bit-wise AND of the loaded multiplicand and bitmask to discard the false multiplicands as a dummy load. In addition, line 3 also computes the bit-wise AND of temporal register s and an inversion of the bitmask. Line 3 then computes the bit-wise OR of the above two AND values and stores the result in s. Thus, at the end of the i-th loop, the register s stores c^i if it is a true multiplicand; otherwise, s preserves the current value. Because all precomputed multiplicands stored in the table are loaded once, this implementation will be resistant to side-channel attacks such as address-featured power/EM attacks, timing-driven cache attacks, and Prime+Probe.

2.3 Existing DL-SCA on RSA (or scalar multiplication)

Several studies have reported the use of DL to estimate the timing of the squaring or multiplication from physical side-channels and thus extract the secret exponent. In [LLQ+20], Lei *et al.* reported that a VGG13-based neural network (NN) can estimate the timing of the squaring and multiplication with a high accuracy from the left-to-right binary exponentiation implemented on a Java card. The target implementation employs a modular squaring/multiplication hardware with power balancing as an SPA countermeasure, which makes it difficult to visually distinguish the difference in shape between squaring and multiplication. Nonetheless, it was shown that the VGG13-based NN can easily distinguish a squaring or multiplication from a power trace, which implies the potential of DL in physical side-channel attacks. In addition, in [CCC+19], Carbone *et al.* showed an attack on an RSA implementation without CRT using square–multiply always exponentiation, along with exponent-, message-, and remainder-blinding. Their attack applied DL to estimate the register address where the multiplication result is stored, which allows for an exponent recovery. In [ZBHV21], Zaid *et al.* improved DL-SCA on RSA (the same implementation as Carbone *et al.*) and ECC by exploiting ensemble learning. Moreover, in [PCBP20], Perin *et al.* showed a non-profiling DL-SCA applied to scalar multiplication. Their method achieved a correct classification through an iterative approach, in which the traces are initially labeled using a clustering-based method, and then repeatedly trained

and re-labeled through the NN itself.

The existing DL-SCAs have mainly focused on any type of *binary* exponentiations (or scalar multiplication) because as they identified the timing of the squaring or multiplication from side-channel trace(s). The applicability of DL to (practical) *windowed* exponentiation is less evaluated in the literature.

In [WPB19, WCPB20], Weissbart *et al.* developed a machine learning based SCA on EdDSA in WolfSSL, which employed a 4-bit fixed-window scalar multiplication (whereas they also targeted another implementation with Montgomery ladder). They demonstrated that a CNN was the most advantageous in attacking the implementation among the tested machine learning techniques, and achieved a full-key recovery of EdDSA. To the best of our knowledge, they conducted the first report on DL-SCA on windowed scalar multiplication. However, note that the WolfSSL implementation performs a dummy load for point addition operand without any countermeasure, which makes the 2^w-classification easier than other open-source implementations considered in this study. In fact, Weissbart *et al.* reported that the signal-to-noise ratio (SNR) of acquired trace is sufficiently high for machine-leaning based SCAs. In addition, there is no report on DL-SCA on RSA–CRT implementation with windowed exponentiation in the literature. A modular multiplication for RSA is implemented differently from a point addition for EdDSA; therefore, the applicability of the existing 2^w-classification-based DL-SCA to RSA is non-trivial.

2.4 RSA–CRT partial key exposure attack

2.4.1 Heninger–Shacham branch-and-prune attack

A partial key exposure attack is an algorithm that recovers the entire RSA–CRT secret key from partial secret key bits, which are typically given by a side-channel attack (*e.g.*, SPA, cache attack, and cold boot attack). The pioneering study of Heninger and Shacham, presented in 2009 [HS09], exploits the fact that a set of RSA–CRT keys satisfies the following relations:

$$ed_p = 1 + k_p(p-1), \tag{1}$$
$$ed_q = 1 + k_q(q-1), \tag{2}$$

where k_p and k_q are integers. The pair of integers (k_p, k_q) take only 2^{16} values in total when the public exponent e is a typical value of $e = 2^{16} + 1$. Because an exhaustive search of 2^{16} candidates is sufficiently feasible, we consider (k_p, k_q) to be known in this paper, as in previous studies. The Heninger–Shacham algorithm uses a leakage model in which the secret key (p, q, d_p, d_q) are known, the remaining bits are unknown, and all known bits are completely correct. Let $a[\lambda]$ be the λ-th bit of an integer a. In the above leakage model, the attacker initially knows that $d_p[\lambda]$ and $d_q[\lambda]$ take a value of either "0," "1," or "unknown" for each λ $(0 \leq \lambda \leq l-1)$. It is known that the Heninger–Shacham algorithm successfully recovers the entire secret key if more-than 50% of the secret key bits are initially obtained [PPS12].

The Heninger–Shacham algorithm estimates the key candidates and discards incorrect key candidates by a branch-and-prune method, from the least significant bit to the upper bits. Let $a^{(\lambda)}$ be the value up-to the $\lambda - 1$th bit of a and let $\tau(a)$ be the consecutive zeros from the least significant bit of a (*i.e.*, the number of trailing zeros). These are formally defined as

$$a^{(\lambda)} = \sum_{j=0}^{\lambda-1} 2^j a[j], \tag{3}$$
$$\tau(a) = \log_2 \gcd(2^l, a). \tag{4}$$

The Heninger–Shacham algorithm exploits the following equations:

$$p[\lambda] + q[\lambda] = (N - p^{(\lambda)}q^{(\lambda)})[\lambda] \bmod 2, \tag{5}$$

$$d_p[\lambda + \tau(k_p)] + p[\lambda] = (k_p(p^{(\lambda)} - 1) + 1 - ed_p^{(\lambda+\tau(k_p))})[\lambda + \tau(k_p)] \bmod 2, \tag{6}$$

$$d_q[\lambda + \tau(k_q)] + q[\lambda] = (k_q(q^{(\lambda)} - 1) + 1 - ed_q^{(\lambda+\tau(k_q))})[\lambda + \tau(k_q)] \bmod 2, \tag{7}$$

which are derived from Eqs. (1) and (2) according to Hensel's lemma. Define Slice$[\lambda]$ as the tuple of the solution of Eqs. (5)–(7) for λ, which is represented by

$$\text{Slice}[\lambda] = (\hat{p}[\lambda], \hat{q}[\lambda], \hat{d_p}[\lambda + \tau(k_p)], \hat{d_q}[\lambda + \tau(k_q)]), \tag{8}$$

where $\hat{d_p}[\lambda]$ is a candidate for $d_p[\lambda]$, represented by a 0 or 1 (which is the same for $\hat{d_q}$, $\hat{p}$, and $\hat{q}$). Note that Slice[0] is determined from public information.

Equations (5)–(7) have two solutions for λ given a sequence of Slice$[\lambda - 1]$, Slice$[\lambda - 2]$, ..., and Slice[0], Therefore, we can construct a binary branch tree whose node at a depth λ is associated with a candidate for Slice$[\lambda]$. We then discard (*i.e.*, prune) the edges/nodes with incorrect candidates that contradict the obtained bits of the secret key. The Heninger–Shacham algorithm constructs and prunes the binary branch tree from Slice[0] to Slice$[l - 1]$ in a depth-first manner.

Let $\bar{d_p}$ and $\bar{d_q}$ be the leakage (*i.e.*, obtained value) of secret exponents, where $\bar{d_p}[\lambda]$ and $\bar{d_q}[\lambda]$ are given by either a 0, 1, or unknown value. Algorithm 3, which employs a stack S, describes the Heninger–Shacham attack which recovers the entire secret key (p, q, d_p, d_q) from $(\bar{d_p}, \bar{d_q})$. In line 3, the root of the binary branch tree Slice[0] is pushed to the stack S because it is determined according to public information. Lines 4-17 constitute the main loop of a depth-first branch-and-prune search. Line 5 pops the parent node from the stack S. Line 11 generates a candidate child nodes with a slice that satisfies Eqs. (5)–(7). Line 13 then examines the consistency of the generated slice with $(\bar{d_p}, \bar{d_q})$. If the slice candidate is inconsistent with $(\bar{d_p}[\lambda], \bar{d_q}[\lambda])$, we discard the candidate (*i.e.*, prune the edge) and move to search other paths; otherwise, we push the candidate to the stack S and continue to search the path. This algorithm terminates and returns the stack (*i.e.*, a path of slices) as the secret key if the most significant bits are determined (*i.e.*, $\lambda = l - 1$).

For the feasibility, the unknown bits in the obtained secret key beforehand should be randomly distributed. The computational and memory complexity increases based on the maximum number of consecutive unknown bits because we should preserve both slice candidates for a parent node until we find the inconsistency. In other words, if there is a long chunk of unknown bits, the brunch-and-prune algorithm will be infeasible. A Coppersmith-type attack should be used if the unknown bits are given as long chunk(s) rather than randomly distributed [Cop97].

2.4.2 Henecka *et al.*'s secret key recovery from noisy leakage

The secret key information obtained by side-channel attacks is not always correct, mainly owing to the noise included in side-channel information (particularly in the case of a power analysis). This means that the obtained secret key bits frequently contain errors, such as a bit flip. The Heninger–Shacham algorithm is inapplicable of achieving such a leakage model because it assumes that the obtained secret key bits are always correct. Henecka *et al.* extended the partial key exposure attack of Heninger and Shacham to recover the RSA–CRT secret key from a partial key containing bit flips at random positions [HMM10]. Let t be a non-negative integer determined for the attack according to the estimated bit flip probability. Henecka *et al.*'s algorithm guesses the Slice branches in a t-bit-wise manner, and then prunes the t-bit-wise candidates that will contain more bit flips than threshold C, which is determined according to the significance level regarding the bit flip probability and t (where the Heninger–Shacham algorithm corresponds to $t = 1$).

Algorithm 3 Heninger–Shacham branch-and-prune algorithm [HS09]

Input: public key (N, e), bit fractions of secret exponents $(\bar{d_p}, \bar{d_q})$, length of secret parameter l, and integer (k_p, k_q)
Output: secret key (p, q, d_p, d_q)

Initialize stack S;
int $\lambda \leftarrow 0$;
Push Slice[0] to S;
while length$(S) > 0$ **do**
  Pop parent_Slice from S;
  $\lambda \leftarrow$ current_bit(parent_Slice); ▷ λ now indicates parent_Slice's bit position
  if $\lambda = l - 1$ **then**
    return Slice[0], Slice[1], ..., Slice[$l - 1$] ▷ Corresponds to (p, q, d_p, d_q)
  end if
  $\lambda \leftarrow \lambda + 1$ ▷ λ now indicates child_Slice's bit position
  Branch parent_Slice to get candidates of child_Slice; ▷ Using Eqs. (5)–(7)
  for all child_Slice candidates **do**
    if $(\bar{d_p}[\lambda], \bar{d_q}[\lambda])$ have no value or are consistent with candidate **then**
      Push Slice[λ] to S;
    end if
  end for
end while

More concretely, Henecka *et al.*'s algorithm considers a node as t-consecutive Slices (*i.e.*, Slice$[\lambda t]$, Slice$[\lambda t + 1], \ldots,$ Slice$[(\lambda + 1)t - 1]$). The algorithm then evaluates the bit inconsistency between the branch node and the obtained secret key (*i.e.*, how many Slice bits are inconsistent with the corresponding obtained bits) to discard incorrect candidates if the number of inconsistent bits exceeds a threshold parameter C corresponding to the significance level. This means that the search is applied while allowing errors below threshold C and preserving such nodes as candidates. However, as with the Heninger–Shacham algorithm, Henecka *et al.*'s algorithm assumes that the bit flips are uniformly distributed. If the obtained secret key contain more or less consecutive bit flips, the true solution may be discarded because it is likely to contain more wrong bits than the threshold C in such a case[4]. In summary, the algorithm will fail if the bit flips are not uniformly distributed. In fact, the bit flips estimated from side-channel information of the windowed exponentiation are not uniformly distributed, as discussed in the following sections.

3 Proposed attack

3.1 Overview

In this section, we present the proposed attack on an RSA–CRT implementation equipped with a windowed exponentiation using a dummy-load based countermeasure. Namely, we present (1) a single-trace power-analysis method to obtain secret exponent(s) through a multiplicand estimation by means of a DL technique and (2) a partial key exposure attack suitable to a partial key leakage obtained from windowed exponentiation through a power analysis.

[4]This indicates that the number of wrong bits in the correct candidates is in the rejection area in statistically testing whether the random bit flip follows the binomial distribution with the bit flip probability. To avoid such a false-positive, we should set the significance level as small as possible (*i.e.*, set t as great as possible); however, it results in an exponential growth of computational cost. Note that, if the assumed bit flip distribution is far different from the true distribution, the confidence interval is no longer meaningful and the results frequently become false.

3.2 Proposed DL-SCA to estimate secret exponents

Difficulty for existing DL-SCAs in attacking windowed exponentiation. Conventional DL-SCA studies on PKC have thus far targeted binary exponentiation [CCC+19, LLQ+20] (or scalar multiplication [PCBP20]). They have basically used an NN to distinguish whether the secret exponent bit is 0 or 1. Their simple extension to windowed exponentiation may be to use a 2^w-classification[5] to identify which the temporal window value is from 0 to 2^{w-1}. However, such a 2^w-classification is more difficult than a 2-classification in practice. Moreover, several open-source windowed implementations have already employed a dummy load scheme to hide the temporal window value in order to prevent such a direct 2^w-classification. To validate our statement, we experimentally confirmed that 2^w-classification with $w = 4$ and 5 only obtained the correct multiplicand with an accuracy of at most 11.53% and 3.624%, respectively[6].

Our basic idea. By contrast, the proposed DL-SCA exploits the dummy load scheme that hides the temporal window value. The proposed attack focuses on the fact that the loading of multiplicands is conducted in a deterministic order from c_0 to c_{2^w-1}. For a temporal window value b, the true load is applied at the b-th loading, and all remaining loads are dummies. This means that the attackers can identify the temporal window value b if they can know the timing of the true load operation from a side-channel trace. From the above idea, we utilize a 2-classification NN that distinguishes true and dummy loads, which resolves the temporal window value estimation into a 2^w times 2-classification problem rather than a 2^w-classification. Such a 2-classification NN will achieve a higher accuracy than 2^w-classification. Moreover, because there is no known countermeasure that protects true and dummy loads from a distinguish attack, the side-channel traces of true and dummy loads will differ from each other in nature. Note that, in the following, we only describe the method used to estimate the temporal window value(s) and not the entire secret exponent. However, the secret exponent can be easily and uniquely reversed from the temporal window values for both sliding- and fixed-window exponentiations. For example, see [GPPT15] for the detailed reversing procedure.

NN training. The proposed DL-SCA is a profiling attack. As with existing profiling DL-SCAs, we first train an NN using a profiling device, which is the same type of device as the target device. We obtain a set of labeled side-channel traces during true and dummy loads from the profiling device and use it to perform a supervised NN training. Let $\beta \in \{0, 1\}$ be the label, which represents the true load and the dummy load with $\beta = 1$ and $\beta = 0$, respectively. Let $q(\beta \mid \mathbf{X}; \hat{\theta})$ be the trained NN, where $\mathbf{X}$ is a random variable of side-channel trace during loading and $\hat{\theta}$ is a trained NN parameter. The trained NN $q(\beta \mid \mathbf{X}; \hat{\theta})$ is an approximation of the conditional probability distribution of true and dummy loads given a side-channel trace of loading $p(\beta \mid \mathbf{X})$; therefore, given a trace $\mathbf{x}$, the trained NN outputs a probability that the loading corresponding to $\mathbf{x}$ is a true load as $q(1 \mid \mathbf{x}; \hat{\theta})$ (and, conversely, the probability for a dummy load as $q(0 \mid \mathbf{x}; \hat{\theta})$). It is worth noting that there is a difference between profiling and target devices; however, a DL technique addressing such a difference has been known [CZLG21].

[5] For the case of sliding-window exponentiation, 2^{w-1}-classification is needed in attacking.

[6] The experimental setup and NN architecture/hyperparameters for 2^w-classification are kept the same as those in used Section 4, excluding the number of outputs at the output layer. Note that the 2-classification implemented in the proposed attack successfully estimated the temporal window values with more-than 99% accuracy using the same NN model; while the NN model suffered from an insufficient model capacity for the 2^w-classification. This result implies that leveraging a 2^w-classification to directly estimate a temporal window value is a more difficult task than the 2-classification. This also indicates that reducing the window value estimation problem to a 2-classification in the proposed attack greatly reduces the development and training costs for an NN model, compared with a 2^w-classification model.

Algorithm 4 Proposed DL-SCA on fixed-windowed exponentiation

Input: Side-channel trace $\bar{\mathbf{x}} = ((\mathbf{x}_i^{(j)})_{i=0}^{2^w-1})_{j=0}^{n-1}$, and trained NN $q(\beta \mid \mathbf{X}; \hat{\theta})$
Output: Sequence of temporal window values $b_0, b_1, \ldots b_{n-1}$
1: **for** $j = 0$ to $n-1$ **do**
2: **int** $b_j \leftarrow \arg\max_i q(1 \mid \mathbf{x}_i^{(j)}; \hat{\theta})$;
3: **end for**
4: **return** $b_0, b_1, \ldots b_{n-1}$;

Temporal window value estimation using profiled NN. Let $\mathbf{x}_i^{(j)}$ denote the side-channel trace for the i-th loading at the j-th multiplication. Suppose that the exponentiation performs n multiplications (excluding squaring) in the main loop ($n = \lfloor l/w \rfloor$ for a fixed-window), and the attacker can know the timings of the multiplications from a side-channel trace. Algorithm 4 describes an attack that estimates temporal window values using a profiled NN $q(\beta \mid \mathbf{X}; \hat{\theta})$. Here, the input is given as a side-channel trace of the entire exponentiation computation (denoted by $\bar{\mathbf{x}} = ((\mathbf{x}_i^{(j)})_{i=0}^{2^w-1})_{j=0}^{n-1}$), which consists of partial traces for each load $\mathbf{x}_i^{(j)}$. Algorithm 4 iteratively estimates the temporal window values in ascending order. Figure 1 is a graphical illustration of the j-th temporal window value estimation in Algorithm 4. For each temporal window value estimation, we apply 2^w inferences for each $\mathbf{x}_i^{(j)}$ ($i \in \{0, 1, \ldots, 2^w - 1\}$) using the trained NN. This indicates that we compute the probability (*i.e.*, confidence rate) for all hypothetical temporal window values being true from the corresponding side-channel trace. We then determine the most likely temporal window value as that with the maximum confidence rate as follows:

$$b_j = \arg\max_i q(1 \mid \mathbf{x}_i^{(j)}; \hat{\theta}), \tag{9}$$

where the NN inference $q(1 \mid \mathbf{x}_i^{(j)}; \hat{\theta})$ corresponds to the confidence rate for the i-th loading at the j-th multiplication[7].

3.3 Proposed RSA–CRT partial key exposure attack

Our new RSA–CRT partial key exposure attack is designed to recover a noisy exponent of windowed exponentiation. (Note here that the attack can be easily extended to a sliding-window by inferring the position and size of temporal windows from a squaring–multiplication sequence like [BBG+17].) Our algorithm assumes that the errors in the estimated temporal window values are uniformly distributed with an error probability. In addition, according to Algorithm 4, the exponent of a fixed-window exponentiation is estimated based on the temporal window value in a w-bit-wise manner. Consequently, the w-bit-wise error(s) exist in an estimated partial exponent value if it is incorrect. In contrast, Henecka *et al.*'s attack model, which assumes that bit flips are uniformly distributed, does not appropriately represent this leakage. To tolerate consecutive bit errors, Henecka's algorithm requires an intentionally large Slice size t or threshold C. This results in an exponential growth in the number of preserved candidates because 2^t child candidates will be branched for a given parent candidate. In addition, if the threshold C is set to a high value, extra 2^t child candidates will occur for each remaining candidate. Hence, Henecka's algorithm will not always be feasible for a (noisy) exponent estimated from temporal windowed values.

Our algorithm introduces a priority-based strategy to branch nodes such that the candidate nodes contain more matches to the obtained secret exponents under the condition

[7] Formally, an arg max function returns a set consisting of augments of maximization. Here, we consider the arg max to return an index integer because the cardinality of $\arg\max_i q(1 \mid \mathbf{x}_i^{(j)}; \hat{\theta})$ is *almost surely* one.

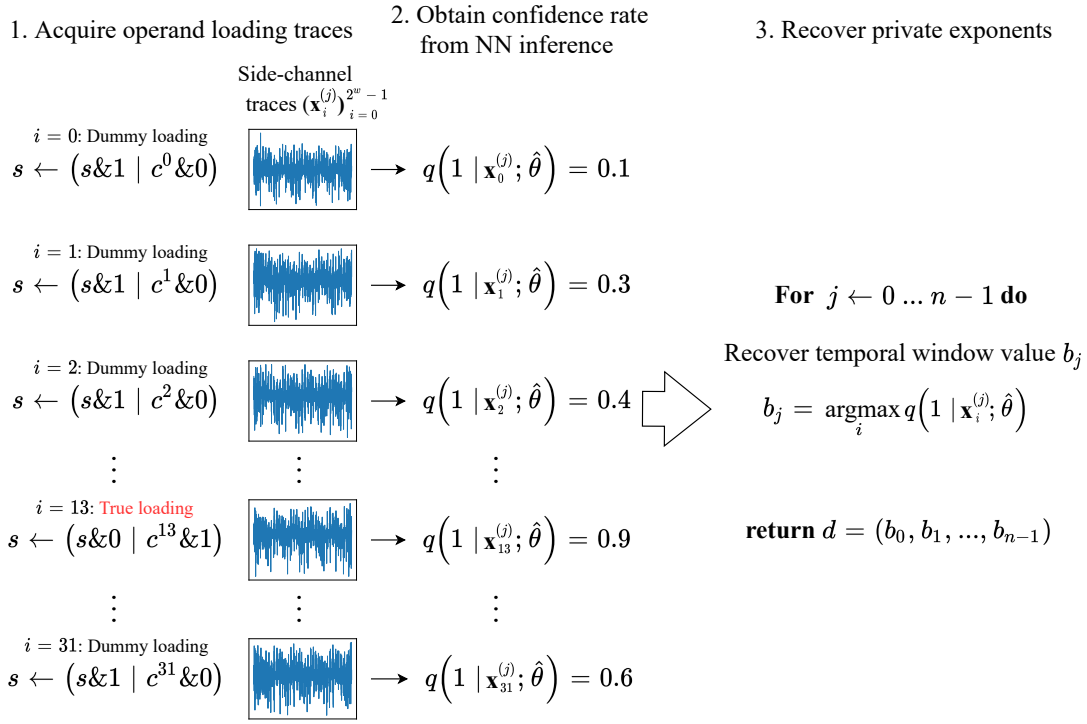

Figure 1: Secret exponent recovery by proposed DL-SCA.

of leakage from a windowed exponentiation. First, we focus on the empirical fact that incorrect child nodes should have fewer matches to the exponent value obtained in advance by SCA. In the conventional depth-first search, all candidates (even that are not likely to be correct) are preserved and examined until they are pruned, regardless of the condition that we can find more likely-correct candidates by a heuristics. Our algorithm prioritizes such likely-correct candidates. Let $x^{(\alpha,\beta)}$ be a chunk from the α-th to β-th bits of variable x and $\text{Slice}^{(0,\lambda-1)}$ be a search path from the least significant bit to the $\lambda-1$-th bit of Slice. That is,

$$x^{(\alpha,\beta)} = (x[\alpha], x[\alpha+1], ..., x[\beta]), \tag{10}$$

$$\text{Slice}^{(0,\lambda-1)} = (\hat{p}^{(0,\lambda-1)}, \hat{q}^{(0,\lambda-1)}, \hat{d_p}^{(0,\lambda-1)}, \hat{d_q}^{(0,\lambda-1)}). \tag{11}$$

First, the basic strategy of our algorithm is to perform branch-and-prune with Eqs. (5)-(7), and then compare candidates and the obtained exponent for each window width w. The priorities of candidates are determined by the comparison results, *i.e.*, the number of window-wise parts providing matches between the obtained exponent value and the candidates. That is, one term in our cost function is simply expressed as the number of w-bit-wise matches between the obtained exponents (*i.e.*, keys) ($\bar{d_p}$ and $\bar{d_q}$) and the candidates ($\hat{d_p}$ and $\hat{d_q}$). Let $I_{\lambda-w}$ be the match value of the $\lambda-w+1$-th w bit from the lower $\lambda-w+1$-th bit to the λ-th bit as follows:

$$I_{\lambda-w} = \delta_{\bar{d_p}^{(\lambda-w+1,\lambda)}, \hat{d_p}^{(\lambda-w+1,\lambda)}} + \delta_{\bar{d_q}^{(\lambda-w+1,\lambda)}, \hat{d_q}^{(\lambda-w+1,\lambda)}}, \tag{12}$$

where δ denotes Kronecker delta. $I_{\lambda-w} = 2$ means that both candidates match with the corresponding obtained exponents. $I_{\lambda-w} = 1$ represents that one of the candidates matches with the corresponding obtained exponent. When $I_{\lambda-w} = 0$, there is no match.

In addition, we introduce the following three conditions for prioritization: (1) to prioritize candidates that match with the obtained key values in the last three w-bit-wise nodes, (2) to penalize (*i.e.*, postpone) candidates that are inconsistent with the obtained key values in the last three w-bit-wise nodes, and (3) to further prioritize candidates that match with the obtained key values in the last four w-bit-wise nodes. More precisely, Condition (1) is satisfied when the three match values $I_{\lambda-w}$, $I_{\lambda-2w}$ and $I_{\lambda-3w}$ for Slice$^{(\lambda-w+1,\lambda)}$, Slice$^{(\lambda-2w+1,\lambda-w)}$, and Slice$^{(\lambda-3w+1,\lambda-2w)}$ are all equal to 2. Conversely, Condition (2) is satisfied when the above three match values are all equal to 0. Condition (3) is satisfied when the four match values $I_{\lambda-w}$, $I_{\lambda-2w}$, $I_{\lambda-3w}$ and $I_{\lambda-4w}$ are all equal to 2. The reason for using at least three match values (*i.e.*the values of current and two prior positions) is that a w-bit-wise error affects two adjacent windows in our algorithm when $\tau(k_p)$ or $\tau(k_q)$ is equal to or more than 1. As a result, in order to evaluate the match/mismatch accuracy between the $2w$-bit-wise candidates and the corresponding obtained keys, we need to evaluate at least successive three comparison results.

We first give the priority to candidates that satisfy Condition (1), and then give the greater priority to the candidate that further satisfy Condition (3) as almost definitely to be correct. On the other hand, if there is no candidate Slice$^{(\lambda-w+1,\lambda)}$ having $I_{\lambda-w} = 2$, Condition (1) is not applicable, and incorrect candidates may continue to be branched during the next two branches for finding Slice$^{(\lambda+1,\lambda+w)}$ and Slice$^{(\lambda+w+1,\lambda+2w)}$. Condition (2) is used to avoid branching incorrect candidates in succession; that is, assuming that the w-bit-wise errors of the obtained key ($\bar{d_p}$ and $\bar{d_q}$) are uniformly and sparsely distributed, we can consider that paths where candidates do not match the obtained key consecutively is unlikely to be correct, and penalize (*i.e.*, postpone) it. From the above ideas, our heuristics defines the cost function of a search path Slice$^{(0,\lambda-1)}$ by C_λ as follows:

$$\begin{aligned} C_0 &= 0, \\ C_\lambda &= C_{\lambda-w} + (2 - I_{\lambda-w}) + h_1 + h_2 + h_3, \end{aligned} \tag{13}$$

where the sub-functions h_1, h_2 and h_3 correspond to Conditions (1),(2) and (3), respectively. These are given as

$$h_1 = \begin{cases} -1 & I_{\lambda-w} = I_{\lambda-2w} = I_{\lambda-3w} = 2 \\ 0 & \text{else} \end{cases}, \tag{14}$$

$$h_2 = \begin{cases} P & I_{\lambda-w} = I_{\lambda-2w} = I_{\lambda-3w} = 0 \\ 0 & \text{else} \end{cases}, \tag{15}$$

$$h_3 = \begin{cases} -P & I_{\lambda-w} = I_{\lambda-2w} = I_{\lambda-3w} = I_{\lambda-4w} = 2 \\ 0 & \text{else} \end{cases}, \tag{16}$$

respectively. Here, P is the cost penalty parameter.

Algorithm 5 shows the proposed partial key exposure attack algorithm with a priority queue Q. In the proposed algorithm, the branch-prune of Slice is performed by grouping together w-bit units (*i.e.*, window size). First, line 2 branches the first w bits, and line 4 calculates the cost given by our heuristics. Here, the branched Slice$^{(0,w-1)}$ are pushed to Q according to their cost C_w. Note that Q always contains the candidate with the lowest cost at the beginning. The main loop starting from line 7 then branches the candidates in a w-bit-wise manner. Line 8 pops the Slice_path (*i.e.*, Slice$^{(0,\lambda-1)}$ if the length is λ) from Q. If the popped Slice_path reaches the most significant bit (*i.e.*, $\lambda = l$), then line 11 returns the resulting path as the recovered key (p, q, d_p, d_q). Otherwise, line 13 branches the following w-bit path from the Slice_path, namely, branches Slice$^{(\lambda,\lambda+w-1)}$. Line 15 then computes the cost $C_{\lambda+w}$ for the candidate Slice$^{(0,\lambda+w-1)}$ and pushes it to Q for the subsequent computation.

Algorithm 5 Proposed heuristic based partial key exposure attack algorithm

Input: Public key (N, e), secret exponents with window-wise error $(\bar{d_p},\bar{d_q})$, length of secret parameter l, and integer (k_p, k_q)
Output: Secret key (p, q, d_p, d_q)

Initialize priority queue Q;
Branch Slice$^{(0,w-1)}$ candidate from Slice[0] to Slice[$w-1$];
for all Slice$^{(0,w-1)}$ candidates **do**
  int $C_w \leftarrow$ calc_cost$(C_0, \bar{d_p}^{(0,w-1)}, \bar{d_q}^{(0,w-1)}, \text{Slice}^{(0,w-1)})$; ▷ According to (13)
  Push Slice$^{(0,w-1)}$ to Q with cost C_w ▷ Note that first element of Q has smallest cost
end for
while true do
  list Slice_path $\leftarrow$ DEQUEUE(Q);
  int $\lambda \leftarrow$ length(Slice_path); ▷ Note that Slice_path expresses Slice$^{(0,\lambda-1)}$
  if $\lambda = l$ **then**
    return Slice_path; ▷ Corresponds to (p, q, d_p, d_q)
  end if
  Branch Slice$^{(\lambda,\lambda+w-1)}$ from Slice_path;
  for all Slice$^{(\lambda,\lambda+w-1)}$ candidates **do**
    $C_{\lambda+w} \leftarrow$ calc_cost$(\bar{d_p}^{(0,\lambda+w-1)}, \bar{d_q}^{(0,\lambda+w-1)}, \text{Slice}^{(0,\lambda+w-1)})$;
    Push Slice$^{(0,\lambda+w-1)}$ to Q with cost $C_{\lambda+w}$;
  end for
end while

Algorithm completeness. The proposed algorithm is clearly complete. Henecka *et al.*'s algorithm employs two conditions for pruning: (i) Eqs. (5)–(7) and (ii) the bit inconsistency between the branched node and the key leakages. Pruning using condition (i) leaves the algorithm complete because any candidate that does not satisfy Eqs. (5)–(7) is clearly not the correct solution. However, pruning using condition (ii) compromises the completeness of the algorithm; here, if the bit flips are concentrated within a certain range of the obtained key, the correct solution in that range will erroneously satisfy the pruning condition, *i.e.*, exceed the threshold C of bit inconsistency with the obtained key. By contrast, the proposed algorithm has completeness because it only uses condition (i) for pruning.

Algorithm complexity. The proposed algorithm applies the following four operations to the branched nodes:

1. Keeps the cost of the branched nodes.
2. Reduces the cost of the branched nodes.
3. Adds the cost of the branched nodes.
4. Penalizes the cost of the branched nodes.

For branched nodes, there may be a node that matches $(\bar{d_p}, \bar{d_q})$. The algorithm applies operation 1 or 2 to the node, and thus the node becomes the parent in the next loop. If there is such a node (*i.e.*a node that matches $(\bar{d_p}, \bar{d_q})$), then the algorithm applies operation 3 or 4 to all other nodes, that is, postpones the nodes[8]. Therefore, if the w-bit-wise errors in $(\bar{d_p}, \bar{d_q})$ are uniformly and sparsely distributed, postponing the incorrect nodes by operations 3 and 4 acts as a pseudo-pruning. Hence, if the number of errors is ϵ, the number

[8]According to [HS09, HMM10], every node branched by an incorrect partial solution includes randomly chosen bits. Under this conjecture, the probability that the incorrect branched node(s) matches with the error of leakage $(\bar{d_p}, \bar{d_q})$ is $1/2^{2w}$ for window width w, which is small enough. Therefore, if one of the branched node matches the $(\bar{d_p}, \bar{d_q})$, the other nodes should be discarded or greatly penalized in practice.

of nodes to be branched can be linearly suppressed to $\mathcal{O}(\epsilon)$[9]. Note that, although our algorithm could suffer from a consecutive error caused during the temporal window value estimation, such phenomenon is unlikely to occur when the errors are uniformly distributed. Namely, although a consecutive error in estimating temporal window values increases the time and memory complexities exponentially by its length, the occurrence probability of consecutive errors is decreased exponentially by its length due to the assumption that errors are uniformly distributed. Hence, the proposed algorithm would be sufficiently feasible if we can achieve a sufficient accuracy in estimating temporal window values. Moreover, our algorithm works even without knowing the maximum number of consecutive errors.

4 Experiment evaluation

4.1 Validation of proposed DL-SCA

4.1.1 Experiment setup

We validate the proposed DL-SCA through an experiment attack conducted on an RSA–CRT implementation constructed using the Gnu MP library. We constructed RSA–CRT software using the modular exponentiation function `mpz_powm_sec` provided by Gnu MP (version 6.1.2)[10], and implemented it on an ARM Cortex-M4 MCU equipped on an STM32F407G-DISC1 board. We set the RSA–CRT key length to 1,024- and 2,048-bits. The window sizes of the fixed-window method are $w = 4$ and $w = 5$ for 1,024- and 2,048-bits, respectively, according to the macro in Gnu MP. We built the Gnu MP library using the `arm-none-eabi-gcc` compiler (version 10-2020-q4-major). We acquired the EM traces using an EM probe Langer EMV-Technik RF-R 50-1 and Keysight DSOS404A oscilloscope with a sampling rate of 400, MSa/s and an amplitude resolution of 8-bits. For NN training, we acquired 61,440,000 EM traces during the true/dummy loading (*i.e.*, line 3 of Algorithm 2) for each key length. For testing the trained NN, we acquired the EM traces of 48 and 100 modular exponentiations with different secret key for 1,024- and 2,048-bit RSA–CRT, respectively.

Regarding the NN architecture, we extended a CNN architecture in ASCAD [BPS+20], which is a common NN architecture in DL-SCA, such that it achieved a sufficiently large model capacity with respect to the computational costs. Then, we conducted multiple experimental attacks using the NN architecture with several different hyperparameters; finally we experimentally determined the hyperparameter values that yield the highest accuracy. Table 2 lists in detail the used NN architectures, which consists of six convolutional layers followed by two fully connected layers. The loss function was the binary cross entropy, learning rate was 0.00005, batch size was 2, 000, and numbers of epochs were 25 and 5 for the 1,024- and 2,048-bit RSA–CRT, respectively. Python 3.7, CUDA 11.0, cuDNN 7.5.1, and Tensorflow 2.4.1 were used to construct the programming environment. Note that the primary purpose of our experiment is to demonstrate the proof-of-concept for the proposed attack and not necessarily to investigate efficient NN architecture(s) for DL-SCA on public-key cryptography. Further investigation for efficient NN architectures in DL-SCAs should be addressed in future studies, as several existing researches address this concern (*e.g.*, [ZBHV21]).

[9] On the other hand, if an error exists in close proximity, the postponement of incorrect nodes does not work as pseudo-pruning, and extra branching of such nodes may be required. In this case, if the number of errors in close proximity is $\epsilon^\star$, then the number of nodes to be branched is $\mathcal{O}(2^{w\epsilon^\star})$.

[10] The algorithm is identical to the latest version 6.2.1.

Table 2: NN architectures.

(a) For 1,024-bit key length.

Layer	Output shape
input	(None, 1000, 1)
normalization	(None, 1000, 1)
conv1d	(None, 1000, 16)
batch_normalization	(None, 1000, 16)
re_lu	(None, 1000, 16)
max_pooling1d	(None, 500, 16)
conv1d	(None, 500, 16)
batch_normalization	(None, 500, 16)
re_lu	(None, 500, 16)
max_pooling1d	(None, 250, 16)
conv1d	(None, 250, 32)
batch_normalization	(None, 250, 32)
re_lu	(None, 250, 32)
max_pooling1d	(None, 125, 32)
conv1d	(None, 125, 32)
batch_normalization	(None, 125, 32)
re_lu	(None, 125, 32)
max_pooling1d	(None, 62, 32)
conv1d	(None, 62, 64)
batch_normalization	(None, 62, 64)
re_lu	(None, 62, 64)
max_pooling1d	(None, 31, 64)
conv1d	(None, 31, 64)
batch_normalization	(None, 31, 64)
re_lu	(None, 31, 64)
global_max_pooling1d	(None, 64)
dense	(None, 128)
re_lu	(None,128)
dense	(None, 64)
re_lu	(None,64)
output	(None,1)

(b) For 2,048-bit key length.

Layer	Output shape
input	(None, 1500, 1)
normalization	(None, 1500, 1)
conv1d	(None, 1500, 32)
batch_normalization	(None, 1500, 32)
re_lu	(None, 1500, 32)
max_pooling1d	(None, 750, 32)
conv1d	(None, 750, 32)
batch_normalization	(None, 750, 32)
re_lu	(None, 750, 32)
max_pooling1d	(None, 375, 32)
conv1d	(None, 375, 64)
batch_normalization	(None, 375, 64)
re_lu	(None, 375, 64)
max_pooling1d	(None, 187, 64)
conv1d	(None, 187, 64)
batch_normalization	(None, 187, 64)
re_lu	(None, 187, 64)
max_pooling1d	(None, 93, 64)
conv1d	(None, 93, 128)
batch_normalization	(None, 93, 128)
re_lu	(None, 93, 128)
max_pooling1d	(None, 46, 128)
conv1d	(None, 46, 128)
batch_normalization	(None, 46, 128)
re_lu	(None, 46, 128)
global_max_pooling1d	(None, 128)
dense	(None, 160)
re_lu	(None,160)
dense	(None, 96)
re_lu	(None,96)
output	(None,1)

4.1.2 Results

First, the NN accuracies to distinguish true and dummy loads from an EM trace were 0.9994 and 0.9966 for 1,024- and 2,048-bit RSA–CRT, respectively. Then, in accord with Eq. (9), we evaluated the accuracy of our temporal window value estimation. As a result, we confirmed that the estimation using the trained NN achieved test accuracies of 99.80% and 99.86% for the 1,024- and 2,048-bit RSA–CRT, respectively. In addition, we also evaluated how many misestimations are included in a single-trace attack (*i.e.*, 128 and 205 temporal window value estimations on one modular exponentiation for the 1,024- and 2,048-bit RSA-CRT, respectively). We confirmed that one single-trace attack contains no more than 2 or 3 errors of temporal window value estimations out of 128 and 205 estimations, respectively. Moreover, 38 single-trace attacks out of 48 contained no estimation errors for the case of the 1,024-bit RSA-CRT. For the 2,048-bit RSA-CRT, 77 single-trace attacks out of 100 contained no estimation errors. The rates of no estimation errors indicate that the success rates of a full secret exponent recovery were 79.17% and 77% for 1,024- and 2,048-bit RSA-CRT, respectively. Table 3 summarizes the results. We confirm from the results that the secret exponents can be estimated with a high accuracy even in the case of 2,048-bit RSA–CRT only when applying the proposed DL-SCA.

Table 3: Estimated results of secret exponents (success rate).

Bit length \ Class	T/D load	Window value	Exponent
1,024-bit	99.94%	99.80%	79.17%
2,048-bit	99.66%	99.86%	77.00%

4.2 Validation of proposed partial key exposure attack

4.2.1 Experiment setup

To evaluate the computational complexity and success rate of our algorithm, we conducted an experimental attack on 50 random secret keys for both 1,024- and 2,048-bit RSA–CRT cases. The generated RSA–CRT secret keys contain error(s) at random position(s), which corresponds to the error in estimating the temporal window values when applying Algorithm 4. Namely, because we evaluate the attack on an Gnu MP fixed-window implementation in this section, an error is given by a w-bit-wise random flip at a random position of multiplication. The algorithm code was written in Python 3.8 and executed on a server with Xeon Gold 6144 and 768 GiB of memory. We set the penalty parameter $P = 10$ for h_2 and h_3 of Algorithm 5 (*i.e.*, Eq.(15) and Eq.(16)).

4.2.2 Results

Figures 2 and 3 are the box plots of the execution time and the queue length for the different numbers of w-bit-wise errors in a logarithmic scale for 1,024- and 2,048-bit CRT–RSA, respectively. We confirmed that a full key recovery is feasible even if 10 and 7 errors are included in the leakage (*i.e.*, exponents estimated using DL-SCA) in the cases of 1,024- and 2,048-bit RSA–CRT, respectively. In addition, the median of computational complexity will grow at a less than exponential rate with the number of errors (note that the vertical axis is in a logarithmic scale). This indicates that the proposed algorithm will efficiently postpone incorrect candidates by exploiting the characteristics of errors included in the obtained key. By contrast, the worst-case complexity grows exponentially with the number of errors. This will occur because the proposed algorithm cannot efficiently postpone incorrect candidates if errors are crowded within a short range. More precisely, for consecutive or close errors, the proposed algorithm requires an almost brute force search for slice candidates over the existent range, which results in an exponential growth in the computational time. In fact, the proposed algorithm assumes that the window width-wise errors are uniformly distributed. If the error distribution has far from a uniform distribution, the proposed algorithm requires a higher complexity, similar to Henecka's algorithm.

Recall that the maximum number of errors included in the estimated exponent described in Section 4.1.2 was no more than two and three for 1,024- and 2,048-bit RSA–CRT, respectively. These results confirm that the combination of the proposed DL-SCA and a partial key recovery algorithm are sufficiently feasible even under the worst case scenario. In summary, we can confirm that the proposed single-trace attack can practically recover a full secret key from state-of-the-art implementations with a windowed exponentiation and side-channel countermeasure (*i.e.*, a dummy load scheme herein).

5 Discussion

5.1 Comparison to conventional major attacks without DL

Big Mac attack is one of the major and pioneering horizontal attacks on RSA, which detects collisions of multiplicands from power traces of multiplications in order to estimate

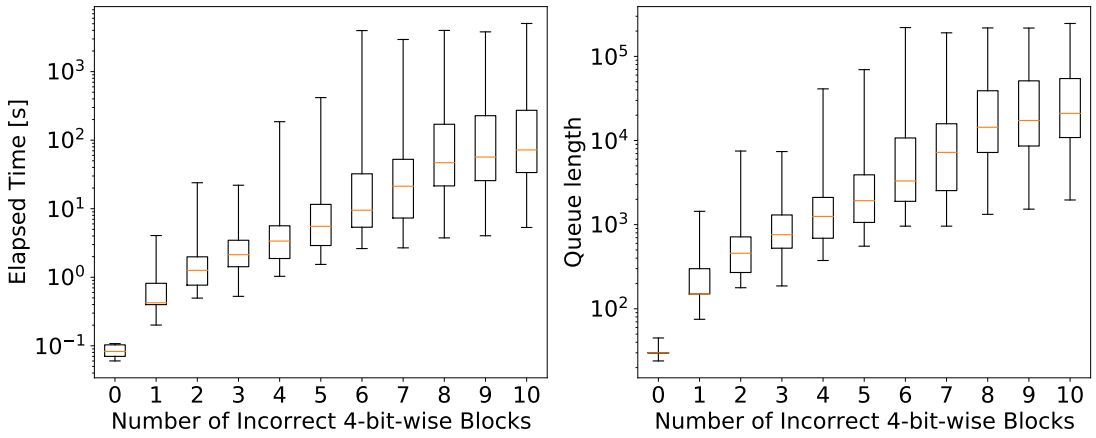

Figure 2: Recovery result for 1,024-bit key.

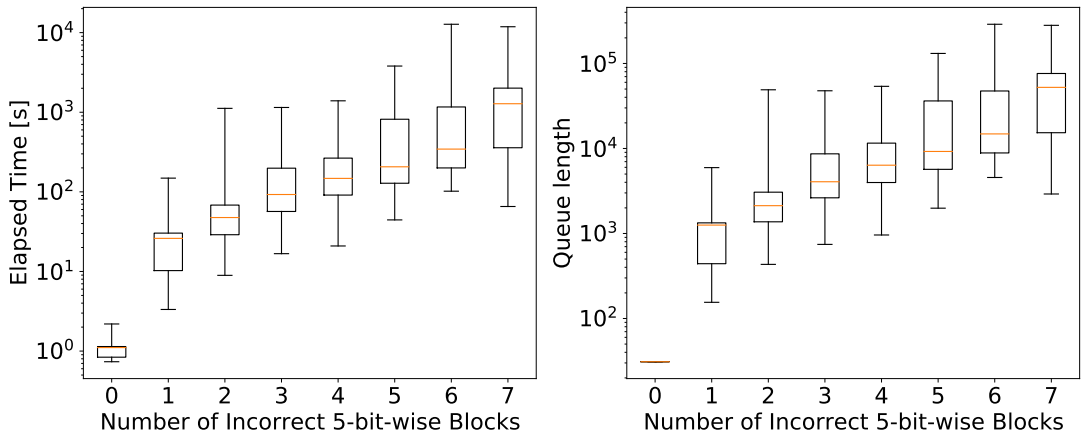

Figure 3: Recovery result for 2,048-bit key.

the secret exponent of sliding window [Wal01]. For the collision detection, Big Mac attack usually averages a set of power consumption traces depending on one of the operands in multi-precision multiplication. Although its validity was confirmed by a simulation in the literature [Wal01], its applicability should always be evaluated with a real device. This is because Big Mac attack is not successful if the noise in power traces cannot be sufficiently removed by averaging. In fact, we experimentally found that the collision detection by Big Mac attack was not successful in our environment.

Template attack is another major profiling attack [CRR02]. It first obtains a conditional probability distribution of side-channel leakage from a given secret value in the profiling phase; then, estimates a target secret value from the obtained likelihood in the attack phase. Here, the conditional probability distribution is estimated assuming that the side-channel leakage follows a multi-dimensional Gaussian distribution. For comparison, we performed a template attack to detect true/dummy load and estimate the temporal window value on the same 1,024-bit RSA implementation as the target described in Section 4. Under the same conditions as the proposed attack, we used 61,440,000 traces for profiling and 98,304 ($= 48 \times 128 \times 16$) traces for the attack (*i.e.*, validation). We determined the point-of-interest

(PoI) as 16 sample points, in which the highest values of absolute difference lied between the average true and dummy load traces[11]. As a result, we confirmed that the template attack distinguished between true and dummy loads with an accuracy of 86.69%. However, the accuracy of temporal window value estimation was only 4.51% and the template attack could never estimate the correct secret exponents. Although the distinguishing between true and dummy loads was sufficiently accurate and successful, the score (*i.e.*, the probability for the true load label being one) for the true load was rarely the highest. Moreover, the estimation accuracy for a temporal window value was as low as a random oracle ($1/16 \approx 6.25\%$), which made the full key recovery unfeasible. This suggests that the leakage model in the template attack does not correctly imitate the true distribution owing to assuming that the noise follows Gaussian distribution. In contrast, our DL-based method does not require such an assumption, and achieves a higher accuracy such that the full key recovery is feasible. Thus, we confirm that our DL-SCA has an advantage against conventional attacks, including template attacks, because of the less strict assumptions about leakage and noise.

5.2 Vulnerability impact on open-source libraries

One of the open-source cryptographic libraries GnuTLS [noa22b] and its background Nettle [noa22c] require Gnu MP for the arbitrary precision arithmetic and thus are directly affected by the Gnu MP vulnerability. In addition, the following libraries adopting a dummy load scheme such as Gnu MP are also considered to have the same type of vulnerability.

OpenSSL. In OpenSSL [noa21b] builds, there is an option to use Gnu MP as a backend, which may be directly affected by the proposed method. The stand-alone version also employs a dummy load scheme using AND with a bitmask and is thus susceptible to similar attacks. Algorithm 6 demonstrates the multiplicand load in OpenSSL implementation, where the operation is equivalent to that of Gnu MP when the window width w is equal to or smaller than 3. This implies that such an OpenSSL implementation can be vulnerable to our attack. However, if $w > 3$, the OpenSSL implementation examines four operands per one load operation at line 16. The proposed method can estimate the timing of a true load but cannot know which operand is stored in the register s among the four candidates. This indicates that the attacker can reduce the number of temporal window value candidates to four (*i.e.*, two-bit) but cannot completely find the true one. The partial key exposure attacks should recover the full key from this partial key information. Recall that Heninger–Shacham-type attacks can feasibly recover the full key if more-than 50% of the bits are exposed [HS09, PPS12]. If $w = 4$ (for 1,024-bit RSA), the attacker can obtain 50% of the secret exponent bits because the proposed method can reduce the temporal window value from four bits to two bits. Therefore, if the attacker can obtain the partial key information with few errors, he/she would succeed in the full key recovery. The high accuracy of our DL-based estimation would make it feasible. However, if $w > 4$ (for 2,048-bit RSA or more), the ratio of exposed bits is insufficient for the partial key exposure attack; thus, the full key recovery would be unfeasible even with the proposed attack.

Botan. Botan [noa21a] will also be vulnerable to the proposed attack because it also employs the dummy load scheme. Algorithm 7 describes the multiplicand load process in Botan, where two true/dummy loads are performed in a single loop. Botan implementation computes the operation corresponding to two loops in the Gnu MP implementation in a single loop. Hence, the number of loops in Botan is half the Gnu MP implementation;

[11] We also performed a t-test that examines the difference of average between true and dummy load traces, and confirmed that the result was consistent with that using the above absolute difference.

Algorithm 6 Operand load in OpenSSL modular exponentiation [noa21b]

Input: Precomputed table $(c^0, c^1, \ldots c^{2^w-1})$, window size w, and window value b
Output: Multiplication operand $s = c^b$

1: **int** $s \leftarrow 0$;
2: **if** $w \leq 3$ **then**
3: **for** $i = 0$ to $2^w - 1$ **do**
4: **int** $s \leftarrow s \mid (c^i \,\&\, (0 - \text{IsEqual}(i, b)))$; ▷ IsEqual$(i, b)$ is 1 if $i = b$, else 0
5: **end for**
6: **return** s;
7: **else**
8: **int** num_loop $\leftarrow 1 \ll (w - 2)$;
9: **int** $j \leftarrow b \gg (w - 2)$;
10: **int** when_fetch $\leftarrow b \,\%$ num_loop;
11: **int** mask0 $\leftarrow 0 - \text{IsEqual}(j, 0)$;
12: **int** mask1 $\leftarrow 0 - \text{IsEqual}(j, 1)$;
13: **int** mask2 $\leftarrow 0 - \text{IsEqual}(j, 2)$;
14: **int** mask3 $\leftarrow 0 - \text{IsEqual}(j, 3)$;
15: **for** $i = 0$ to num_loop $- 1$ **do**
16: $s \leftarrow s \mid (((c^{i+0\cdot\text{num\_loop}} \,\&\, \text{mask0}) \mid$
$(c^{i+1\cdot\text{num\_loop}} \,\&\, \text{mask1}) \mid$
$(c^{i+2\cdot\text{num\_loop}} \,\&\, \text{mask2}) \mid$
$(c^{i+3\cdot\text{num\_loop}} \,\&\, \text{mask3})) \,\&\, (0 - \text{IsEqual}(i, \text{when\_fetch})))$;
17: **end for**
18: **return** s;
19: **end if**

however, the number of true/dummy loads conducted in Botan is still identical to that of Gnu MP in total. Therefore, there is a definite possibility to estimate the temporal window values by monitoring all processes executed in lines 5 and 6, and identifying the true load among them using the proposed method. To validate our approach, we implemented Algorithm 7 with Gnu MP code, and conducted an experimental true/dummy load detection test on the implementation. The experimental setup is the same as that of 1,024-bit RSA implementation described in Section 4. We used 2,048,000 traces for NN training and 102,400 traces for the validation. The NN was trained up-to 50 epochs. We confirmed that the trained NN achieved the highest accuracy of 99.69% in distinguishing between true and dummy loads at 32 epochs. This result validates that the proposed attack can also be applied to Botan implementations.

Algorithm 7 Operand load in Botan modular exponentiation [noa21a]

Input: Precomputed table $(c^0, c^1, \ldots c^{2^w-1})$, window size w, and window value b
Output: Multiplication operand $s = c^b$

1: **int** $s \leftarrow 0$;
2: **for** i from 0, 2, 4, ..., to $2^w - 2$ **do**
3: **int** mask0 $\leftarrow$ MakeBitMask(i, b);
4: **int** mask1 $\leftarrow$ MakeBitMask$(i + 1, b)$;
▷ MakeBitMask(i, b) returns $(11\ldots1)_{(2)}$ if $i = b$, else 0
5: $s \leftarrow s \mid (c^i \& \text{mask0})$;
6: $s \leftarrow s \mid (c^{i+1} \& \text{mask1})$;
7: **end for**
8: **return** s;

Algorithm 8 Operand load in Libgcrypt modular exponentiation [Pro21]

Input: Precomputed table $(c^1, c^3, \ldots c^{2^w-1})$, window size w, and window value b
Output: Multiplication operand $s = c^b$
1: **int** $s \leftarrow 0$;
2: **for** i from 1, 3, 5 to $2^w - 1$ **do**
3: mask $\leftarrow$ MakeBitMask(i, b); ▷ MakeBitMask(i, b) returns $(11\ldots1)_{(2)}$ if $i = b$, else 0
4: $s \leftarrow (s\&\neg\text{mask}) \mid (c^i\&\text{mask})$;
5: **end for**
6: **return** s;

Libgcrypt. Libgcrypt [Pro21] employs sliding-window exponentiation. The multiplicand is also loaded from a precomputation table such as the aforementioned fixed-window table, and the side-channel attacker who obtained the temporal window value and the squaring–multiplication sequence can recover the secret exponent. Algorithm 8 describes the operand load process in Libgcrypt. We can see that Libgcrypt also employs a dummy load scheme using a bitmask. More precisely, the value of the load destination resister/memory value s shown in line 4 is rewritten only when $i = b$, that is, the true load. This can be exploited by the proposed attack, similar to the case of Gnu MP. In addition, Libgcrypt currently employs an exponent blinding to counter Bernstein *et al.*'s cache attack [BBG+17]. However, the proposed attack can still recover the secret exponent because the exponent blinding can be broken using the Euclidean algorithm if an attacker obtains three completely correct blinded exponents. The proposed DL-SCA is sufficiently accurate to obtain such blinded exponents, as evaluated in Section 4.

5.3 Why and how proposed DL-SCA works

Possible/potential leakage source. The 2-classification NN of the proposed attack can distinguish true and dummy loads with an extremely high accuracy. An important question here is how the NN distinguishes them, or what difference the NN exploits. More concretely, what is the difference in side-channel traces during true and dummy loads? One major possibility would be whether the register/memory preserves the previous value or stores a different value from the previous state. When loading a multiplicand, the register/memory value is rewritten only when a true load is executed; the register/memory preserves the previous value in cases of dummy loads. This suggests that the true load may result in a meaningfully different power trace from those of the dummy loads, and the NN may distinguish true and dummy loads owing to this difference.

Treating imbalanced data problem. It has been reported that DL-SCA may suffer from the imbalanced data problem that inherently exists in machine learning applications. In [PHJ+19], Picek *et al.* demonstrated that the imbalanced data problem would negatively affect DL-SCA on AES and presented how the so-called synthetic minority oversampling technique (SMOTE) can be used to address the imbalanced data problem. In [ISUH21], Ito *et al.* further analyzed the impact and causes of the imbalanced data problem in DL-SCA on AES. They demonstrated an analytical solution to solve the problem. However, when attacking the windowed-exponentiation of RSA–CRT by distinguishing a true/dummy load, the imbalanced data problem would inevitably occur because the frequency of a true load is far fewer than that of a dummy load. (Namely, the frequency of a true load is $1/(2^w - 1)$ of that of a dummy load.) A major negative effect on such imbalanced data problems is that the NN output probability would be biased by the distribution of labels [ISUH21]. In this case, a dummy load is likely to have a higher probability than a true load because the dataset contains more samples of the dummy load label than those of the true load. Thus, our 2-classification NN may outputs a probability distribution where the dummy load

probability is higher and true load probability is lower than the true probability, owing to the bias incurred by the imbalanced data. However, the temporal window value estimation in the proposed attack is not influenced by this bias because we estimate the temporal window value as the *arg max* of probabilities for being true load. Even if the NN output probability is biased, the arg max of the probabilities is invariant to such bias. Accordingly, we can confirm that the effectiveness of the proposed temporal window value estimation using a 2-classification NN. Note here that our solution outperforms the SMOTE-based solution because the training dataset in our method is not compromised by low-quality samples synthesized like SMOTE. In addition, Ito *et al.*'s solution cannot be applied directly because it is specific for DL-SCA on symmetric ciphers.

Impact on the value of w. We discuss the impact of the maximum window size w on the attack. In the temporal window value estimation using a 2-classification NN, the accuracy of 2-classification using an NN is assumed to be invariant to the value of w, because it only focuses on the load operation, which is independent of w. In fact, Table 3 depicts that the proposed attack achieves almost the same accuracy in estimating the temporal window value for 1,024-bit and 2,048-bit RSAs, which use $w = 4$ and 5, respectively. (The accuracy for the 2,048-bit RSA is slightly higher than that for the 1,024-bit RSA in the experiment.) Thus, we can use an NN model independently of w, which indicates the proposed attack is scalable to the value of w. In contrast, the value of w has an impact on the computational cost of the partial key exposure attack algorithm. An error in estimating the temporal window value causes a w-bit consecutive error. If w is large, a w-bit-wise branch-and-prune should be performed, which yields larger computational and memory costs. However, the value of w usually is not large in practice because a larger value of w leads to a larger pre-computation cost. Considering both pre-computation and main loop costs standpoints, the practical and optimal value of w is said to be less than eight [Koç95]. The proposed algorithm can achieve key recovery from 1,024-bit and 2,048-bit RSA–CRT with $w = 4$ and 5, respectively, even if the number of temporal value errors is greater than the experimentally confirmed value.

Leveraging our attack in cross-device scenario. We used an identical device for both training (*i.e.*, profiling) and testing (*i.e.*, attack) phases. In a real attack, the attacker usually targets a device that cannot be used for training because the training commonly requires a total control over devices[12]. This indicates that the accuracies of distinguishing between dummy and true loads and estimating temporal window value degrade owing to the difference between leakage characteristics of profiled and target devices. In [CZLG21], Cao *et al.* reported that a cross-device DL-SCA, which indicates that the attacker targeted a device different from a profiling device, sometimes achieved non-negligibly lower performance or failed the key recovery. Cao *et al.* presented a solution based on domain adaption to this problem, which mitigated the performance degradation and achieved a successful key recovery in a cross-device setting. Thus, although this study used an identical device for both profiling and target phases for the proof-of-concept, our attack would be practical even in a real cross-device scenario thanks to Cao *et al.*'s method. Further evaluation in a cross-device setting should be conducted in future studies.

5.4 Countermeasure

The proposed method exploits the two following facts to recover the secret exponent (*i.e.*, temporal window value): (i) the timing of the true load is determined in a deterministic manner, whereas it directly corresponds to the temporal window value, and (ii) the

[12]In contrast, in this scenario, the attacker does not require the control of the target device, and it is sufficient for attacker only to trigger the decryption to acquire side-channel trace.

Algorithm 9 Proposed operand loading process

Input: Precomputed table $(c^0, c^1, \ldots c^{2^w-1})$, Window size w, Window value b
Output: Multiplication operand $s = c^b$

1: **int** $r_{\text{value}} \leftarrow \text{GenerateRandom}_l()$; ▷ Generate an l-bit random mask.
2: **int** $s \leftarrow c^0 \oplus c^1 \oplus \cdots \oplus c^{2^w-1} \oplus r_{\text{value}}$;
3: **int** $r_{\text{order}} \leftarrow \text{GenerateRandom}_w()$; ▷ Generate a w-bit random mask.
4: **int** $b_{\text{masked}} \leftarrow b \oplus r_{\text{order}}$;
5: **for** i from 0 to $2^w - 1$ **do**
6: **int** mask $\leftarrow \text{MakeBitMask}(i, b_{\text{masked}})$;
▷ $\text{MakeBitMask}(i, b_{\text{masked}})$ returns $11\ldots1$ if $i = b_{\text{masked}}$, else 0
7: $s \leftarrow s \oplus (c^{i \oplus r_{\text{order}}} \& \neg \text{mask} \mid r_{\text{value}} \& \text{mask})$;
8: **end for**
9: **return** s;

register/memory value is rewritten only when the true load is executed, as discussed in Section 5.3. To mitigate the proposed attack, we introduce two randomization techniques: load order randomization[13] and dummy load value randomization.

Algorithm 9 describes the proposed randomized multiplicand loading. To randomize the order of the multiplicand load, line 4 masks the temporal window value using a w-bit random number r_{order} as b_{masked}. In the For loop, index i is compared to b_{masked} instead of b, and is unmasked as $i \oplus b_{\text{masked}}$ under the (dummy) loading of the multiplicand. Although this implementation correctly applies the true load only if $i = b_{\text{masked}}$ (and a dummy load otherwise), the order is randomized owing to the random mask r_{order}.

In addition, to randomize the dummy load value, line 1 also generates a l-bit random mask r_{value}, and initializes the temporal register s using r_{value} as $c^0 \oplus c^1 \oplus \cdots \oplus c^{2^w-1} \oplus r_{\text{value}}$. Line 7 then subtracts $c^{i \oplus r_{\text{order}}}$ from s as $s \leftarrow s \oplus c^{i \oplus r_{\text{order}}}$ in the dummy load, whereas it subtracts r_{value} as $s \leftarrow s \oplus r_{\text{value}}$ in the true load. Thus, because all other multiplicands and the mask are subtracted from s, only the true multiplicand c^b remains in the register s at the end of the For loop. In this implementation, s is written whenever any (true or dummy) load operation is performed. Thus, Algorithm 9 will mitigate the two vulnerabilities, which the proposed attack exploited.

6 Conclusion

In this paper, we proposed a new DL-SCA method and a partial key exposure attack applicable to an RSA–CRT software implementation with an application to the modular exponentiation of Gnu MP. The proposed DL-SCA aims at distinguishing between true and dummy loads of the multiplicand under the windowed method for recovering the value of the exponent with high accuracy. Because the multiplicand loading from precomputed tables (with a dummy load) is commonly used for some open-source *windowed* exponentiation, we mentioned that the proposed attack can also be applicable to some open-source implementations. We also proposed a new partial key exposure attack suitable for the leakage model of the windowed exponentiation. The proposed partial key exposure attack overcomes the limitations of the conventional algorithm in [HMM10] by using the heuristics and the number of matches with the secret exponents obtained. We also presented a countermeasure against the proposed attack, which can mitigate the two major vulnerabilities that the proposed attack exploited. Further detailed investigations regarding the applicability/extension of the proposed attack to other major implementations of public key cryptography remain as areas of future study.

[13] It is similar to a conditional swap (cswap) randamization in [ABW+21] and is its generalization.

Acknowledgments

This research was supported by JST CREST Grant No. JPMJCR19K5 and the JSPS KAKENHI Grant No. 21H04867 and No. 20K19765, Japan.

References

[ABW+21] Monjur Alam, Yilmaz Baki, Frank Werner, Niels Samwel, Alenka Zajic, Daniel Genkin, Yuval Yarom, and Milos Prvulovic. Nonce@Once: A single-trace em side channel attack on several constant-time elliptic curve implementations in mobile platforms. In *IEEE European Symposium on Security and Privacy*, 2021.

[AG01] Mehdi-Laurent Akkar and Christophe Giraud. An Implementation of DES and AES, Secure against Some Attacks. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, Lecture Notes in Computer Science, pages 309–318, Berlin, Heidelberg, 2001. Springer.

[AKD+18] Monjur Alam, Haider Adnan Khan, Moumita Dey, Nishith Sinha, R. Callan, A. Zajić, and M. Prvulović. One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA. In *USENIX Security Symposium*, 2018.

[BBG+17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding Right into Disaster: Left-to-Right Sliding Windows Leak. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, Lecture Notes in Computer Science, pages 555–576, Cham, 2017. Springer International Publishing.

[BJPW13] Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, and Justine Wild. Horizontal and Vertical Side-Channel Attacks against Secure RSA Implementations. volume 7779, 2013.

[BPS+20] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ASCAD database. *Journal of Cryptographic Engineering*, 10(2):163–188, June 2020.

[CCC+19] Mathieu Carbone, Vincent Conin, Marie-Angela Cornélie, François Dassance, Guillaume Dufresne, Cécile Dumas, Emmanuel Prouff, and Alexandre Venelli. Deep Learning to Evaluate Secure RSA Implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 132–161, February 2019.

[CFG+10] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. Horizontal Correlation Analysis on Exponentiation. In Miguel Soriano, Sihan Qing, and Javier López, editors, *Information and Communications Security*, Lecture Notes in Computer Science, pages 46–61, Berlin, Heidelberg, 2010. Springer.

[CK09] Jean-Sébastien Coron and Ilya Kizhvatov. An Efficient Method for Random Delay Generation in Embedded Software. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, Lecture Notes in Computer Science, pages 156–170, Berlin, Heidelberg, 2009. Springer.

[Cop97] Don Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 10(4):233–260, September 1997.

[CRR02] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, LNCS, pages 13–28, 2002.

[CZLG21] Pei Cao, Chi Zhang, Xiangjun Lu, and Dawu Gu. Cross-Device Profiled Side-Channel Attack with Unsupervised Domain Adaptation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 27–56, August 2021.

[DBN+01] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. Advanced Encryption Standard (AES). November 2001. Last Modified: 2021-03-01T01:03-05:00.

[DLLM15] Ibrahima Diop, Pierre-Yvan Liardet, Yanis Linge, and Philippe Maurine. Collision based attacks in practice. In *Euromicro conference on Digital System Design*, pages 367–374, 2015.

[GPPT15] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation. In *Cryptographic Hardware and Embedded Systems – CHES 2015*, volume 9293, pages 207–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. Series Title: Lecture Notes in Computer Science.

[Gra] Torbjorn Granlund. Defeating modexp side-channel attacks with data-independent execution traces. page 9.

[HMA+10] Naofumi Homma, Atsushi Miyamoto, Takafumi Aoki, Akashi Satoh, and Adi Samir. Comparative Power Analysis of Modular Exponentiation Algorithms. *IEEE Transactions on Computers*, 59(6):795–807, June 2010. Conference Name: IEEE Transactions on Computers.

[HMM10] Wilko Henecka, Alexander May, and Alexander Meurer. Correcting Errors in RSA Private Keys. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, Lecture Notes in Computer Science, pages 351–369, Berlin, Heidelberg, 2010. Springer.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA Private Keys from Random Key Bits. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, Lecture Notes in Computer Science, pages 1–17, Berlin, Heidelberg, 2009. Springer.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. 2008.

[IIT02] Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. Address-bit differential power analysis of cryptographic schemes OK-ECDH and OK-ECDSA. volume 2523, pages 129–143, 2002.

[ISUH21] Akira Ito, Kotaro Saito, Rei Ueno, and Naofumi Homma. Imbalanced data problems in deep learning-based side-channel attacks: Analysis and solution. *IEEE Transactions on Information Forensics and Security*, pages 1–1, 2021.

[KDKL17] Ievgen Kabin, Zoya Dyka, Dan Kreiser, and Peter Langendoerfer. Horizontal address-bit DPA against montgomery kP implementation. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, December 2017.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019. ISSN: 2375-1207.

[Koç95] Cetin Kaya Koç. Analysis of sliding window techniques for exponentiation. *Computers & Mathematics with Applications*, 30(10):17–24, 1995.

[LH20a] JongHyeok Lee and Dong-Guk Han. DLDDO: Deep learning to detect dummy operations. In *International Conference on Information Security Applications (WISA)*, LNTCS, pages 73–85, 2020.

[LH20b] JongHyeok Lee and Dong-Guk Han. Security analysis on dummy based side-channel countermeasures—case study: AES with dummy and shuffling. *Applied Soft Computings*, 93:314–328, 2020.

[LLQ+20] Qi Lei, Chao Li, Kexin Qiao, Zhe Ma, and Bo Yang. VGG-Based Side Channel Attack on RSA Implementation. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1157–1161, December 2020. ISSN: 2324-9013.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015. ISSN: 2375-1207.

[MD99] Thomas S. Messerges and Ezzy A. Dabbish. Investigations of Power Analysis Attacks on Smartcards. 1999.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and E. Prouff. Breaking Cryptographic Implementations Using Deep Learning Techniques. *SPACE*, 2016.

[noa21a] Botan: Crypto and TLS for Modern C++ — Botan, December 2021. https://botan.randombit.net/.

[noa21b] OpenSSL Cryptography and SSL/TLS Toolkit, December 2021. https://www.openssl.org/.

[noa22a] The GNU MP Bignum Library, January 2022. https://gmplib.org/.

[noa22b] The GnuTLS Transport Layer Security Library, January 2022. https://www.gnutls.org/.

[noa22c] Nettle - a low-level crypto library, January 2022. https://www.lysator.liu.se/~nisse/nettle/.

[PCBP20] Guilherme Perin, Łukasz Chmielewski, Lejla Batina, and Stjepan Picek. Keep it Unsupervised: Horizontal Attacks Meet Deep Learning. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 343–372, December 2020.

[PHJ+19] Stjepan Picek, Annelie Heuser, Alan Jovic, Shivam Bhasin, and Francesco Regazzoni. The Curse of Class Imbalance and Conflicting Metrics with Machine Learning for Side-channel Evaluations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, (1):209–237, 2019.

[PPS12] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A Coding-Theoretic Approach to Recovering Noisy RSA Keys. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, Lecture Notes in Computer Science, pages 386–403, Berlin, Heidelberg, 2012. Springer.

[Pro21] The People of the GnuPG Project. Libgcrypt, February 2021. Publisher: The GnuPG Project.

[UXT+22] Rei Ueno, Keita Xagawa, Yutaro Tanaka, Akira Ito, Junko Takahashi, and Naofumi Homma. Curse of Re-encryption: A Generic Power/EM Analysis on Post-Quantum KEMs. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 296–322, 2022.

[Wal01] C. D. Walter. Sliding Windows Succumbs to Big Mac Attack. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, Lecture Notes in Computer Science, pages 286–299, Berlin, Heidelberg, 2001. Springer.

[WCPB20] Léo Weissbart, Łukasz Chmielewski, Stjepan Picek, and Lejla Batina. Systematic side-channel analysis of curve25519 with machine learning. *Journal of Hardware and System Security*, (4):314–328, 2020.

[WPB19] Léo Weissbart, Stjepan Picek, and Lejla Batina. One trace is all it takes: Machine learning-based side-channel attack on EdDSA. In *International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE)*, LNTCS, pages 86–105, 2019.

[YF14] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, June 2017.

[ZBHV21] Gabriel Zaid, Lilian Bossuet, Amaury Habrard, and Alexandre Venelli. Efficiency through diversity in ensemble models applied to side-channel attacks – a case study on public-key algorithms –. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, (3):60–96, 2021.