

Second Round Key Recovery and a Floating-Point Attack on DNN-based AES Implementations

Sisung Kim, Minjae Lee, and Dongjae Lee*

Kangwon National University, Chuncheon, Republic of Korea
{sisung13,dlalswo1030,dongjae.lee}@kangwon.ac.kr

Abstract. At EUROCRYPT 2026, G rault et al. introduced natural DNN implementations of block ciphers: ReLU networks that agree exactly with the underlying cipher on binary inputs but extend it continuously to real-valued inputs. They showed that this extension exposes the first round key to recovery, and proposed a generic Secure Blackbox Transformation that turns an arbitrary DNN-based implementation into a provably secure implementation. We revisit both sides of their result. On the attack side, the known attacks on the unprotected implementation reach only the first round key. Under the same threat model, we refine the absorption property of the natural S-box and combine it with a chosen-plaintext collision test, validated empirically, to recover the second round key of AES-256 in about 2^{39} queries. On the defense side, their transformation is provably secure over the reals, but that guarantee does not carry over to finite precision, where any neural-network implementation must ultimately run. We give a floating-point attack that recovers the first round key of the protected AES-128 implementation in 128 chosen queries, driven by an input value that the defense fails to round to a bit. Such a value exists in `bfloat16`, `float16`, `float32`, and `float64`, so moving to higher precision does not remove the weakness. We confirm the recovery in all four formats. Finally, we block the attack with a clamp placed before the defense’s rounding step, which keeps every finite input in $[0, 1]$ in the working precision and changes neither the cipher nor the exact-real security guarantee.

Keywords: Deep neural cryptography · AES · cryptanalysis · key recovery · floating-point arithmetic · finite-precision security.

1 Introduction

Deep learning has become one of the most consequential technologies of the past decade, reshaping many areas of science and society, and cryptography is no exception. In 2002, Klimov, Mityagin, and Shamir analyzed and broke a neural-network key-exchange scheme [12]. As deep learning rose to prominence, Gohr showed in 2019 that a deep neural network can be trained as a distinguisher

* Corresponding author.

and used as the core of a competitive key-recovery attack on round-reduced Speck32/64 [8]. This opened an active line of work that interprets and extends such neural distinguishers [1,2,5,18,19]. In the reverse direction, cryptanalytic techniques have been applied to neural networks themselves, extracting the parameters of a trained ReLU network [4], and later doing so in polynomial time under assumptions of oracle access, known architecture, and exact arithmetic [3].

At EUROCRYPT 2026, G  rault et al. introduced a new direction in this interplay: implementing cryptographic primitives as ReLU-based deep neural networks [7]. They laid the foundations of this area by defining correctness and security for such implementations. Their construction, the *natural implementation*, applies to any such primitive. It is assembled from *corner functions*: ReLU gadgets that express any Boolean function by pinning its value at each binary input, building on the expressiveness of rectifier networks [16]. The network thereby matches the primitive exactly on binary inputs and extends it continuously to real-valued ones.

The natural implementation, however, comes with a structural weakness. Although a legitimate user submits only binary plaintexts, an adversary may submit arbitrary real values. G  rault et al. exploit this with two attacks that recover the first round key. The first uses the symmetry of the natural implementation of XOR. It perturbs a plaintext bit by a small real value and recovers the first round key bit by bit using a collision test. The second targets the first-round S-boxes. At $\text{SBox}^{-1}(0\text{x}00)$, the unique input whose S-box output is zero, the corner active at that input carries a zero output coefficient. A small perturbation of all its bits toward the interior of $[0, 1]$ therefore leaves the output unchanged. The perturbation is said to be *absorbed*. This absorption lets the attacker detect when an S-box input equals $\text{SBox}^{-1}(0\text{x}00)$ and recover the key byte that was XORed into it.

As a defense against these attacks, G  rault et al. [7] introduce a generic *Secure Blackbox Transformation* that turns a possibly insecure DNN-based implementation into a secure one, using two ReLU-based parts. A STEP function sanitizes the inputs and outputs, rounding each real value toward 0 or 1. A RECT function then catches the unsafe middle range that STEP leaves non-binary, and a masking layer built from it zeroes the output whenever any input falls in that range. They prove a *Perfect Simulation* theorem: oracle access to the transformed implementation reveals no information about the secret key beyond what black-box access to the underlying binary primitive already provides.

1.1 Our Contributions

We revisit the natural implementation of AES and its provably secure defense, and make three contributions. Table 1 summarizes the two attacks and their costs.

- **Recovery of the second round key on the unprotected implementation.** We push the S-box absorption technique one round deeper. Once the first round key is recovered, round 1 is a known bijection on binary

Table 1. Summary of our attacks. A query is a single evaluation of the network on one chosen plaintext.

Target	Recovered	Cost
Unprotected AES-192	Full master key (K_0 , half of K_1)	$\approx 2^{38}$ queries
Unprotected AES-256	Full master key ($K_0 \ K_1$)	$\approx 2^{39}$ queries
Protected AES-128	First round key (K_0)	128 queries

states. This lets us drive the state entering the second key addition to any chosen binary value. We recover the second round key one column (four bytes) at a time. A small perturbation of a single plaintext bit propagates through round 1 and, via MixColumns, spreads to the four bytes of one column entering the second key addition. We characterize the perturbation that round 1 delivers to the four round-2 S-boxes and give a criterion for when it is absorbed. The condition we target is that the four bytes affected by the perturbation all equal $\text{SBox}^{-1}(0x00)$ when they enter the round-2 S-boxes. We also explain why wrong candidates almost never collide, and validate the attack with exhaustive censuses.

- **A floating-point break of the provably secure defense.** We show that the defense does not survive finite precision. In `bfloat16`, `float16`, `float32`, and `float64`, we find a representable value that STEP leaves non-binary and that RECT fails to catch. Feeding this value in place of one plaintext bit produces one of two clearly distinguishable outputs, according to whether the corresponding key bit is 0 or 1. We derive the value and its effect on the first key addition exactly from the rounding rules. We then confirm experimentally, on both networks of the unmodified reference code, that the two outputs are distinguishable. In this way, 128 queries recover the first round key of the protected AES-128.
- **A robust defense.** We close the gap by placing a clamp to $[0, 1]$, built from two ReLUs, before STEP. G rault et al. [7] define the same clamp, but their realization can leave a large input outside $[0, 1]$ after rounding. Ours keeps every finite input inside $[0, 1]$ in the working precision (Lemma 1), and changes neither the cipher nor the Perfect Simulation proof. This is an attack-specific hardening: the clamp guards the input only, so a trigger arising inside the network would escape it (Section 5.2).

Finite-precision gaps in provably secure constructions. Our floating-point break does not contradict the Perfect Simulation theorem (Theorem 1), which holds over the reals. Rather, it is an instance of a broader class of gaps between exact-real security guarantees and their finite-precision implementations. Mironov [14] showed that the Laplace mechanism of differential privacy, provably private over the reals, leaks the exact database through the low-order bits of its floating-point samples, and repaired it by a rounding-robust sampler. Jin et al. [10] later extended such attacks to deployed differential-privacy systems.

In lattice cryptography, Lin et al. [13] showed that the floating-point error of the Falcon sampler is itself key-dependent.

Threat model. We assume the black-box model of [7]: the adversary submits chosen real-valued plaintexts and observes the outputs returned by the implementation, without access to the weights, activations, internal state, or key. For the exact-real analysis, these outputs are real-valued. In the finite-precision setting of Section 4, the adversary observes the values returned in the working floating-point format, including non-finite values such as NaN or infinity when they occur.

1.2 Organization

Section 2 introduces the preliminaries, including the natural implementation and the Secure Blackbox Transformation of [7]. Section 3 presents a method to recover the second round key of the unprotected natural implementation. We validate the method by recovering the master key of AES-256. Section 4 develops the floating-point attack, a 128-query key recovery, with an experimental evaluation across formats and networks. Section 5 gives a rounding-robust input clamp that blocks this floating-point trigger. Section 6 concludes this paper.

2 Preliminaries

2.1 Natural Implementation of AES

Gérault et al. [7] define the *natural implementation* of a Boolean function by extending its Boolean behavior to real-valued inputs using ReLU-based corner functions. By composing these natural implementations, they construct the natural implementation of AES, which we denote by NN_{AES} when referring to the network as a formal function.

AES round structure. AES operates on a 128-bit state represented as a 4×4 matrix of bytes [15,6]. Encryption begins with an initial AddRoundKey (ARK), which XORs the state with a round key. Each full round then applies SubBytes (SB), ShiftRows (SR), MixColumns (MC), and ARK. The final round omits MC. As in [7], we assume that the key schedule is evaluated in advance and that the resulting round keys are embedded in the network.

Throughout this paper, we use zero-based round-key indexing: K_0 denotes the key embedded in the initial AddRoundKey operation, and K_1 denotes the key embedded in the following AddRoundKey operation.

The natural implementation of AES replaces ARK, SB, and MC by their respective natural implementations, while SR remains the fixed byte permutation of standard AES.

Corner functions and sums of corners. Let

$$\text{ReLU}(x) = \max\{0, x\}.$$

For a Boolean vertex $b = (b_0, \dots, b_{\kappa-1}) \in \{0, 1\}^\kappa$, define

$$\text{corner}_b(x) = \text{ReLU}\left(\sum_{i:b_i=1} x_i + \sum_{i:b_i=0} (1 - x_i) - \kappa + 1\right).$$

At Boolean inputs,

$$\text{corner}_b(b) = 1, \quad \text{corner}_b(x) = 0 \quad \text{for every } x \in \{0, 1\}^\kappa \setminus \{b\}.$$

For a Boolean function $f : \{0, 1\}^\kappa \rightarrow \{0, 1\}$, let

$$C = \{b \in \{0, 1\}^\kappa : f(b) = 1\}.$$

The corresponding sum of corners is

$$\Sigma_C(x) = \sum_{b \in C} \text{corner}_b(x),$$

and the natural implementation of f is

$$\text{NN}_f(x) := \Sigma_C(x).$$

Hence,

$$\text{NN}_f(x) = f(x) \quad \text{for every } x \in \{0, 1\}^\kappa.$$

For real-valued inputs, several corner functions may be simultaneously active, yielding a continuous piecewise-linear extension of the Boolean function.

Natural implementations of XOR and AddRoundKey. In the natural implementation of AES, AddRoundKey is implemented bitwise by the natural implementation of XOR. For two inputs,

$$\begin{aligned} \text{NN}_{\text{XOR}}(x_1, x_2) &= \text{ReLU}(x_2 - x_1) + \text{ReLU}(x_1 - x_2) \\ &= |x_1 - x_2|. \end{aligned}$$

Thus, for a Boolean round-key bit $K_r[i] \in \{0, 1\}$,

$$\text{NN}_{\text{XOR}}(x_i, K_r[i]) = |x_i - K_r[i]|.$$

For a state $x = (x_0, \dots, x_{127})$ and a binary round key K_r , the natural implementation of AddRoundKey is defined coordinate-wise by

$$\text{NN}_{\text{ARK}}(x, K_r)_i := \text{NN}_{\text{XOR}}(x_i, K_r[i]) = |x_i - K_r[i]|, \quad i = 0, \dots, 127.$$

Natural implementations of the S-box and MixColumns. Identifying each AES S-box output with its 8-bit vector, the natural S-box implementation is

$$\text{NN}_{\text{SB}}(x) = \sum_{b \in \{0,1\}^8} \text{corner}_b(x) \cdot \text{SBox}(b),$$

where $\text{SBox}(b) \in \{0,1\}^8$ denotes the bit-vector representation of the S-box output. Equivalently, this is the vector-valued form of the scalar sum-of-corners construction of [7].

MixColumns requires the XOR of four bytes. Its bitwise four-input XOR is implemented by eight parallel natural implementations of the four-input parity function, one for each bit position. Formally, at bit position j ,

$$\begin{aligned} \text{NN}_{\text{XOR}_8}(a, b, c, d)_j \\ = \sum_{\substack{u \in \{0,1\}^4 \\ u_0 \oplus u_1 \oplus u_2 \oplus u_3 = 1}} \text{corner}_u(a_j, b_j, c_j, d_j). \end{aligned}$$

Thus, this four-input XOR is the natural implementation of the four-input parity function itself rather than an iteration of the two-input natural implementation of XOR. The two constructions agree on Boolean inputs but can differ on real-valued inputs.

The multiplications by 0x02 and 0x03 used in MixColumns are treated analogously and replaced by their natural implementations. For one input column (a_0, b_0, c_0, d_0) , the formal network computation is

$$\begin{aligned} a_1 &= \text{NN}_{\text{XOR}_8}(\text{NN}_{\text{MUL}_2}(a_0), c_0, d_0, \text{NN}_{\text{MUL}_3}(b_0)), \\ b_1 &= \text{NN}_{\text{XOR}_8}(\text{NN}_{\text{MUL}_2}(b_0), a_0, d_0, \text{NN}_{\text{MUL}_3}(c_0)), \\ c_1 &= \text{NN}_{\text{XOR}_8}(\text{NN}_{\text{MUL}_2}(c_0), a_0, b_0, \text{NN}_{\text{MUL}_3}(d_0)), \\ d_1 &= \text{NN}_{\text{XOR}_8}(\text{NN}_{\text{MUL}_2}(d_0), b_0, c_0, \text{NN}_{\text{MUL}_3}(a_0)). \end{aligned}$$

The coefficient-0x01 terms are passed directly to the four-input XOR, matching the construction of [7].

The complete natural implementation of AES composes these operations according to the AES round structure. By construction,

$$\text{NN}_{\text{AES},k}(p) = \text{AES}_k(p) \quad \text{for every binary plaintext } p.$$

The released code of G erault et al. provides two such networks. The *base network* is the direct composition just described, with SubBytes and the MixColumns coefficients as separate natural blocks. The *T-tables network* instead combines the S-box with the coefficients 0x02 and 0x03 into single corner blocks on the byte entering SubBytes. The base network and the T-tables network agree on binary inputs, but they need not agree on real-valued inputs. We analyze the second round key recovery of Section 3 separately on the base network and the T-tables network, since it depends on how the S-box and the coefficients are split into blocks. The floating-point attack of Section 4 touches only the first key addition, which the two networks share, and we run it against both.

2.2 Secure Blackbox Transformation

Gérault et al. [7] introduce the *Secure Blackbox Transformation* for protecting a DNN implementation against real-valued queries. Let

$$B(k, \cdot) : \{0, 1\}^n \rightarrow \{0, 1\}^m$$

denote a keyed binary cryptographic primitive on n -bit inputs and m -bit outputs, and let

$$D(k, \cdot) : \mathbb{R}^n \rightarrow \mathbb{R}^m$$

be a DNN implementation satisfying

$$D(k, p) = B(k, p) \quad \text{for every } p \in \{0, 1\}^n.$$

The transformation sanitizes inputs and outputs with the function STEP_ε , its *sanitizer*, defined by

$$\text{STEP}_\varepsilon(x) = \frac{\text{ReLU}(x - \varepsilon) - \text{ReLU}(x - (1 - \varepsilon))}{1 - 2\varepsilon}.$$

Over exact real arithmetic,

$$\text{STEP}_\varepsilon(x) = \begin{cases} 0, & x \leq \varepsilon, \\ \frac{x - \varepsilon}{1 - 2\varepsilon}, & \varepsilon < x < 1 - \varepsilon, \\ 1, & x \geq 1 - \varepsilon. \end{cases}$$

The rectangular function RECT_ε detects the middle range $(\varepsilon, 1 - \varepsilon)$ that STEP_ε leaves non-binary. It is

$$\text{RECT}_\varepsilon(x) = \frac{1}{\varepsilon} (\text{ReLU}(x - 1) + \text{ReLU}(x) - \text{ReLU}(x - \varepsilon) - \text{ReLU}(x - (1 - \varepsilon))),$$

with

$$\text{RECT}_\varepsilon(x) = \begin{cases} 0, & x \leq 0, \\ \frac{x}{\varepsilon}, & 0 < x < \varepsilon, \\ 1, & \varepsilon \leq x \leq 1 - \varepsilon, \\ \frac{1 - x}{\varepsilon}, & 1 - \varepsilon < x < 1, \\ 0, & x \geq 1. \end{cases}$$

Summing it over the coordinates gives the mask that the transformation uses to zero the output whenever any coordinate falls in that range. For $p = (p_0, \dots, p_{n-1}) \in \mathbb{R}^n$, define

$$\text{MASK}_\varepsilon(p) = \sum_{i=0}^{n-1} \text{RECT}_\varepsilon(p_i).$$

Following [7], we call an input *safe* if every coordinate satisfies $p_i \leq \varepsilon$ or $p_i \geq 1 - \varepsilon$, and *unsafe* otherwise. A safe input is binarized cleanly, $\text{STEP}_\varepsilon(p) \in \{0, 1\}^n$, whereas an unsafe input has $\text{MASK}_\varepsilon(p) \geq 1$.

The protected implementation, shown in Figure 1, is

$$D_S(k, p) = \text{ReLU}(\text{STEP}_\varepsilon(D(k, \text{STEP}_\varepsilon(p))) - \text{MASK}_\varepsilon(p)),$$

where STEP_ε and the final ReLU are applied coordinate-wise and the scalar mask is subtracted from every output coordinate.

For binary inputs,

$$\text{STEP}_\varepsilon(p) = p, \quad \text{MASK}_\varepsilon(p) = 0,$$

and therefore

$$D_S(k, p) = D(k, p) = B(k, p), \quad p \in \{0, 1\}^n.$$

For unsafe inputs, the output-side STEP_ε bounds every output coordinate by 1, while $\text{MASK}_\varepsilon(p) \geq 1$. Hence the final ReLU produces the all-zero output.

Perfect Simulation. The security of the Secure Blackbox Transformation is guaranteed by Theorem 1 over exact real arithmetic.

Theorem 1 (Perfect Simulation, paraphrased from [7]). *Assume that $D(k, p) = B(k, p)$ for every binary input $p \in \{0, 1\}^n$. Any probabilistic polynomial-time adversary with oracle access to D_S can be perfectly simulated by a probabilistic polynomial-time adversary with oracle access only to B .*

For a safe query, the simulator queries B on the binary value $\text{STEP}_\varepsilon(p)$ and reproduces the public sanitization and masking operations. For an unsafe query, it returns the all-zero output.

Released implementation. G  rault et al. [7] describe STEP_ε with $\varepsilon = 1/3$ and apply it to both the input and the output. However, the released natural AES code implements it with $\varepsilon = 1/4$ hard-coded and applies it only to the input. We take the released realization with $\varepsilon = 1/4$ as our reference and, unless stated otherwise, also add the output-side STEP, so our target is D_S of Figure 1. Section 4.4 also evaluates the attack at other values of ε , including $1/3$.

3 Second Round Key Recovery and Full AES-192/256

The attacks of G  rault et al. [7] recover the first round key K_0 of the natural AES. In this section, we show how to recover the second round key K_1 as well.

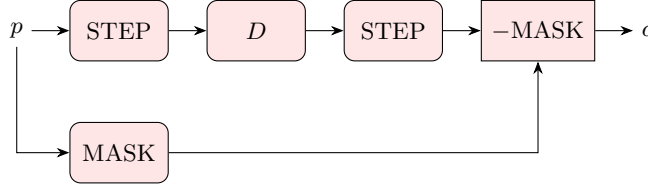


Fig. 1. The protected natural AES pipeline $D_S(p) = \text{ReLU}(\text{STEP}(D(\text{STEP}(p))) - \text{MASK}(p))$: an input STEP sanitizes p , the network D computes the primitive, an output-side STEP re-sanitizes, and MASK(p) forces an all-zero response on unsafe inputs.

3.1 S-box Absorption at 0x52

We first analyze a local property of the natural S-box at

$$0\text{x}52 = \text{SBox}^{-1}(0\text{x}00).$$

Moving a bit of a binary input *inward* by $\eta \in (0, \frac{1}{2})$ means

$$0 \mapsto \eta, \quad 1 \mapsto 1 - \eta.$$

Gérault et al. [7] observed that at 0x52 a small inward perturbation of all eight bits of an S-box input leaves the output unchanged (the perturbation is *absorbed*). They used this property to test whether a first-round S-box input equals 0x52, and thereby recovered the first round key. We analyze the property in more detail and use it to attack the second round key. Appendix A.1 makes it precise (Lemma 2 and Corollary 1). Let v be an S-box input and move $n \geq 1$ of its eight bits inward by a common $\eta < \frac{1}{2}$. If $v = 0\text{x}52$, the output is unchanged if and only if $n \geq 2$. If $v \neq 0\text{x}52$, the output changes for every n . We build an attack using this property. Here and in Appendix A, n counts perturbed bits of one byte and is unrelated to the input length n of Section 2.2.

3.2 Recovering the Second Round Key

Once K_0 is known, the first natural AES round is a known bijection on binary states. The adversary can therefore realize any binary state M_1 immediately before the second NN_{ARK} from the chosen binary plaintext

$$P = \text{SB}^{-1}(\text{SR}^{-1}(\text{MC}^{-1}(M_1))) \oplus K_0.$$

We recover K_1 independently by state column. For a column index $c \in \{0, 1, 2, 3\}$, let $K_{1,c}$ denote the corresponding four-byte column of K_1 , and let g denote the c -th column of the internal state M_1 . If g is binary, the c -th column after the second NN_{ARK} is $g \oplus K_{1,c}$. Our goal is to find the unique state column

$$g^* = K_{1,c} \oplus (0\text{x}52, 0\text{x}52, 0\text{x}52, 0\text{x}52)$$

for which all four bytes of this column equal the absorbing input $0x52$. Once g^* is identified, the key column follows as $K_{1,c} = g^* \oplus (0x52, 0x52, 0x52, 0x52)$.

We therefore scan all 2^{32} values of the state column and call each of them a *candidate* for g^* . For a candidate g , we choose a state M_1 whose c -th column equals g , with the other three columns held at the same fixed value for every candidate, and realize it through the plaintext P above. It remains to distinguish g^* from the other $2^{32} - 1$ candidates.

Although any binary M_1 can be realized in this way, the perturbation cannot be applied to this internal state directly. It must instead be introduced in the plaintext and allowed to propagate through the first natural AES round. Since SR^{-1} maps the c -th column onto the c -th diagonal, the four bytes of P at positions $(r, (c + r) \bmod 4)$, for $r = 0, \dots, 3$, are the only bytes that depend on g .

For a candidate g , we find a plaintext P_g that realizes it. We construct one real-valued probe for each of the 32 bits of P_g on the c -th diagonal by moving that bit inward by η and leaving all other bits unchanged. Each candidate is thus tested with one binary baseline P_g and 32 real-valued probes.

For the correct candidate g^* , the four target bytes after the second NN_{ARK} all equal $0x52$, so the perturbation carried by a probe enters four natural S-boxes at $0x52$. When the perturbation is absorbed at all four S-boxes, the perturbed plaintext and the baseline produce the same ciphertext. For a wrong candidate, at least one target byte differs from $0x52$, and Lemma 2 gives no such guarantee.

We write $\text{bits}(P)$ for the 128-bit binary encoding of a plaintext P and $\text{round}(\cdot)$ for the coordinate-wise rounding of the oracle’s real-valued output to the nearest integer, and say that two queries *collide* when their rounded outputs agree. For a candidate g , the collision score N_g is the number of its 32 probes that collide with the baseline P_g . We distinguish g^* from the wrong candidates by this score.

Appendix A follows a probe through round 1 and determines, over the reals, when it collides by absorption, which yields Proposition 1.

Proposition 1 (Scores of the collision test). *Fix the base network, exact real arithmetic, and $0 < \eta < \frac{1}{16}$. Call a collision forced if it follows from absorption at the four round-2 S-boxes. Counting forced collisions, for a column whose key column $K_{1,c}$ is uniformly random:*

- (i) *the score of the correct candidate is the sum of four independent terms, each equal to 0, 7, or 8 with probability $\frac{16}{256}$, $\frac{46}{256}$, and $\frac{194}{256}$ respectively. In particular, $\Pr[N_{g^*} < 8] = \frac{25}{131072} \approx 1.9 \times 10^{-4}$;*
- (ii) *every wrong candidate scores 0, 7, or 14. At most 1024 among the $2^{32} - 1$ wrong candidates score 7 or 14. A wrong candidate scoring 14 exists, depending on the key column, with probability $1 - (255/256)^4 \approx 1.6\%$.*

We accept a candidate g when its collision score reaches a threshold τ , that is, when $N_g \geq \tau$. By Proposition 1, the choice $\tau = 8$ admits the correct candidate except with probability about 1.9×10^{-4} per column, and admits no wrong candidate other than the rare ones scoring 14. The final re-encryption check of Algorithm 1 rejects a candidate scoring 14 if one is admitted. Algorithm 1

summarizes the complete procedure, the recovery of K_0 followed by the column-wise recovery of K_1 . Section 3.3 tests the proposition on the network itself, in floating-point arithmetic.

Complexity. Scanning one column costs $33 \cdot 2^{32}$ chosen-plaintext queries, one baseline and 32 probes per candidate, and the preceding recovery of K_0 is negligible in comparison. The AES-192 master key consists of K_0 and the first 64 bits of K_1 , so two columns suffice and the attack costs $2 \cdot 33 \cdot 2^{32} \approx 2^{38}$ queries. The AES-256 master key is $K_0 \| K_1$, so all four columns are needed and the attack costs $4 \cdot 33 \cdot 2^{32} \approx 2^{39}$ queries. The non-target columns of M_1 are fixed to 0 throughout, since the scores of Proposition 1 do not depend on them (Appendix A.2). We validate on AES-256 only, since AES-192 uses the same procedure on two of the four columns.

Algorithm 1: Full master-key recovery for the unprotected natural AES-256 (K_0 then K_1)

Input: Oracle $E(\cdot) = \text{NN}_{\text{AES}}(\cdot) \in \mathbb{R}^{128}$; perturbation η and threshold τ ; an observed pair (P^*, C^*) with $C^* = \text{round}(E(P^*))$

Output: Master key $K_0 \| K_1$, or FAIL

/ Phase 1: recover K_0 bit by bit from the XOR symmetry */*

draw 4 random base plaintexts $b \in \{0, 1\}^{128}$

for $i \leftarrow 0$ **to** 127 **do**

 let $v_b^\pm$ be b with coordinate i set to $\pm\eta$

$K_0[i] \leftarrow 1$ **if** $\text{round}(E(v_b^+)) \neq \text{round}(E(v_b^-))$ **for some base** b , **else** 0

/ Phase 2: scan each column in full, record every g with $N_g \geq \tau$ */*

for $col \leftarrow 0$ **to** 3 **do**

$O \leftarrow \mathbf{0}$ (fixed non-target state); $A_{col} \leftarrow \emptyset$

for $g \leftarrow 0$ **to** $2^{32} - 1$ **do**

$M_1 \leftarrow O$ with column col set to the 4 bytes of g (big-endian)

$P \leftarrow \text{SB}^{-1}(\text{SR}^{-1}(\text{MC}^{-1}(M_1))) \oplus K_0$; $c_0 \leftarrow \text{round}(E(\text{bits}(P)))$

$N_g \leftarrow$ number of the 32 probes v of P with $\text{round}(E(v)) = c_0$

if $N_g \geq \tau$ **then** $A_{col} \leftarrow A_{col} \cup \{g\}$

if $A_{col} = \emptyset$ **then return** FAIL

/ Phase 3: select the combination that reproduces (P^*, C^*) */*

foreach $(g_0, g_1, g_2, g_3) \in A_0 \times A_1 \times A_2 \times A_3$ **do**

$K_1 \leftarrow$ the columns g_{col} , each byte $\oplus 0x52$

if re-encrypting P^* under $K_0 \| K_1$ reproduces C^* **then return** $K_0 \| K_1$

return FAIL

3.3 Experimental Validation

We validate the attack against the reference code of G  rault et al. [7], imported unmodified. The attack queries it only as an oracle. To build the AES-256 target,

we reuse its key-independent gadgets for the key schedule and the 14 rounds, and we check the result against the FIPS-197 test vector. We report three kinds of experiment. A *margin census* evaluates, for many random keys, the correct candidate of every column together with a random sample of wrong candidates. An *enumeration* counts, for the same keys, the wrong candidates that Proposition 1 predicts to score 14. A *full census* runs the attack on one key and scans the whole 2^{32} space of every column.

Instantiation. We use $\eta = 0.05$ for both the K_0 symmetry test and the K_1 probes, and $\tau = 8$ following Proposition 1. Appendix B discusses the choice of η and alternative values of τ . The K_0 test uses four random base plaintexts because a single one occasionally masks a bit through a cancellation later in the network. All collisions are read from the rounded final ciphertext only. All seeds and logs are available in our public repository (Appendix D).

Margin census. To isolate the K_1 collision test, the margin census assumes K_0 is already known and takes it directly from the key schedule rather than recovering it in Phase 1. It runs on CPU in `float32` and `float16`. In `float32`, it covers 10,000 random keys, that is, 40,000 columns. For each column, it scores the correct candidate and 10^3 random wrong candidates, 4×10^7 wrong candidates in all. In `float16`, it covers the same 10,000 keys with the same sampling.

For the wrong candidates, the largest sampled score is 0 in 39,987 columns and 7 in the other 13. No other value occurs. Proposition 1(ii) predicts 9.5 such hits at score 7, in good agreement. In `float16`, the largest sampled score is 0 in 39,987 columns and 7 in the other 13, the same 13 columns as in `float32`.

For the correct candidate, the score ranges from 6 to 32 with median 31 (Figure 2). It falls below $\tau = 8$ in 10 columns, against 7.6 expected from Proposition 1(i). Algorithm 1 loses a column only when the correct candidate’s score falls below the threshold, so $\tau = 8$ recovers all four K_1 columns for 9,990 of the 10,000 keys, a rate of 99.9%. The remaining 10 keys are handled by the ranking fallback of Appendix B, since their correct candidates score 6 or 7 and at most 1024 wrong candidates score above 0. In `float16`, the correct candidate scores 5 to 32 with median 31 and falls below τ in 10 columns, again a rate of 99.9% (Appendix A.4).

Enumeration of the candidates scoring 14. The probability of about 1.6% in Proposition 1(ii) is the probability that at least one of the 2^{32} candidates of a column scores 14. Since the margin census samples only 10^3 wrong candidates per column, it does not meet any of them. We therefore determine, as in Appendix A.3, the candidates scoring 14 for the 10,000 keys of the margin census. Of the 40,000 columns, we find that 636 have one such candidate and 3 have two, 642 candidates in all. The 639 columns are 1.60% of the total, in agreement with the prediction.

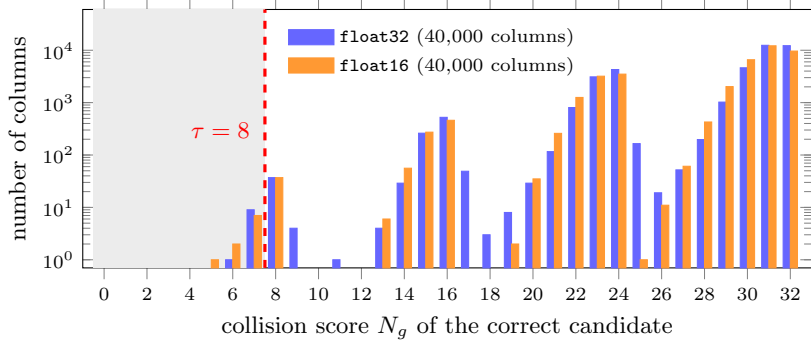


Fig. 2. Collision score of the correct candidate over the 40,000 columns of the margin census, in `float32` and in `float16` (log scale). Scores 22–32 account for 97.3% and 97.2% of the columns, and 10 columns (0.025%) fall below $\tau = 8$ (shaded) in each format.

Full censuses. We run two kinds of full census, querying the oracle in `float16` on two RTX 5080 GPUs. Scanning one column takes about 70 hours, so a full key takes about 140 hours on the two GPUs.

The first kind runs Algorithm 1 as is, with $\tau = 8$, and records every candidate that reaches the threshold. We run it on two random keys. On both, the attack recovers the correct first and second round keys, which we verified against an observed plaintext–ciphertext pair. Over the full $8 \cdot 2^{32}$ candidate space, no wrong candidate scores above 7.

The second kind records the score of every candidate. We run it on a third random key, which the attack recovers as well. In each of its four columns, every wrong candidate scores either 0 or 7, and exactly 1024 score 7. This is the structure of Proposition 1(ii), and the 1024 are precisely the colliding wrong candidates that Appendix A.3 enumerates from the key alone. The correct candidates score 24, 32, 28, and 31, against 24, 32, 30, and 31 predicted by Appendix A.2, the deficit of 2 being the rounding loss discussed in Appendix A.4. No other score occurs, so every collision among these 2^{34} candidates is one that our analysis predicts.

Full census on the T-tables network. On the T-tables network, Appendices A.3 and A.4 predict that every wrong candidate scores 0 and that the correct candidate scores at least 20. We run the second kind of full census on this network, on a fourth random key. The attack recovers the key, in about 110 hours since this network is smaller. Over all $4 \cdot 2^{32}$ candidates, every wrong candidate scores 0. The correct candidates score 31, 28, 30, and 32 against the predicted 31, 30, 30, and 32. On CPU in `float32`, a sample of 64 of the 1024 wrong candidates per column that score 7 on the base network scores 0 on this network, as predicted.

4 A Floating-Point Attack on the Protected Implementation

In this section, we identify an implementation-level vulnerability in the defense of G erault et al. [7] that is caused by finite precision. We use it to recover the full first round key from the protected AES-128 implementation with 128 chosen queries.

4.1 Binary Floating-Point Arithmetic

We consider radix-2 floating-point formats and assume round to nearest, ties to even, as standardized by IEEE 754 [9]. Let F be a binary floating-point format with f stored fraction bits. A finite normalized value can be written as

$$x = (-1)^\sigma \left(1 + \sum_{i=0}^{f-1} b_i 2^{-(i+1)} \right) 2^e = (-1)^\sigma (1.b_0b_1 \cdots b_{f-1})_2 \times 2^e, \quad b_i \in \{0, 1\},$$

where $\sigma \in \{0, 1\}$ is the sign bit and e the unbiased exponent. Table 2 summarizes the parameters for the binary formats considered in our experiments. For a fixed unbiased exponent e , positive normalized values lie in the interval

$$2^e \leq x < 2^{e+1}.$$

Within this interval, adjacent representable values are separated by

$$\Delta_F(e) = 2^{e-f}, \quad x \in [2^e, 2^{e+1}).$$

The spacing therefore doubles with each unit increase of the exponent, so the distance between adjacent representable values grows with their magnitude.

For a real value z , let $\text{fl}(z)$ denote the result of rounding z to the working format. Under round to nearest, ties to even, a value lying exactly halfway between two adjacent representable values is rounded to the endpoint whose significand has an even least-significant bit.

4.2 Numerical Analysis of the Floating-Point Discrepancy

We now show that at $\varepsilon = 1/4$, the value hard-coded in the released code (Section 2.2), the floating-point evaluation of STEP departs from its exact-real value at a representable input. For this parameter, the input transformation is computed as

$$\text{STEP}_{1/4}(x) = \frac{\text{ReLU}(x - \frac{1}{4}) - \text{ReLU}(x - \frac{3}{4})}{\frac{1}{2}}.$$

For $x \geq 3/4$, both ReLU arguments are nonnegative and their difference is exactly $1/2$. Therefore,

$$\text{STEP}_{1/4}(x) = 1.$$

However, this equality is not always preserved by floating-point computation. Consider the representable input

$$t_F = 2^{f-1} + 1,$$

which we call the *trigger* of the format. Since

$$2^{f-1} \leq t_F < 2^f,$$

the unbiased exponent in this range is $e = f - 1$. The spacing between consecutive representable values around t_F is therefore

$$2^{e-f} = 2^{(f-1)-f} = \frac{1}{2}.$$

The two subtractions in $\text{STEP}_{1/4}$ have the exact values

$$t_F - \frac{1}{4} \quad \text{and} \quad t_F - \frac{3}{4}.$$

Since the spacing here is $1/2$, both lie halfway between two adjacent representable values. Round to nearest, ties to even, selects the neighbor whose last significand bit is 0, which is t_F for the first value and $t_F - 1$ for the second:

$$\text{fl}\left(t_F - \frac{1}{4}\right) = t_F, \quad \text{fl}\left(t_F - \frac{3}{4}\right) = t_F - 1.$$

Thus, although their difference over the reals is

$$\left(t_F - \frac{1}{4}\right) - \left(t_F - \frac{3}{4}\right) = \frac{1}{2},$$

the floating-point implementation computes

$$t_F - (t_F - 1) = 1.$$

Consequently,

$$\widehat{\text{STEP}}_{1/4}(t_F) = \frac{1}{1/2} = 2,$$

where $\widehat{\text{STEP}}$ denotes the value produced by the floating-point implementation. We keep the hat wherever the computed value has to be contrasted with the exact-real one. In the experimental sections, where every quantity is a computed one, we drop it and write $\text{STEP}(t)$ for the value the implementation returns at the trigger.

The mask, the other part of the defense, does not intervene. The scalar function building it,

$$\begin{aligned} \text{RECT}_{1/4}(x) = & 4(\text{ReLU}(x - 1) + \text{ReLU}(x) \\ & - \text{ReLU}(x - \tfrac{1}{4}) - \text{ReLU}(x - \tfrac{3}{4})), \end{aligned}$$

Table 2. Parameters of the binary floating-point formats considered in this work, and the resulting per-format trigger $t_F = 2^{f-1} + 1$.

Format	Fraction bits f	Exponent bits	Trigger t_F
bfloat16	7	8	65
float16	10	5	513
float32	23	8	4,194,305
float64	52	11	$2^{51} + 1$

vanishes for every $x \geq 1$, so $\text{RECT}_{1/4}(t_F) = 0$ over the reals. The finite-precision evaluation gives the same value, since $\text{ReLU}(x)$ and $\text{ReLU}(x - \frac{1}{4})$ both round to t_F while $\text{ReLU}(x - 1)$ and $\text{ReLU}(x - \frac{3}{4})$ both round to $t_F - 1$. The finite-precision discrepancy is therefore confined to $\widehat{\text{STEP}}$:

$$\boxed{\widehat{\text{STEP}}_{1/4}(t_F) = 2, \quad \widehat{\text{RECT}}_{1/4}(t_F) = 0.}$$

Since $t_F \geq 1 - \varepsilon$, the trigger is a safe input in the sense of Section 2.2. Over the reals, STEP_ε maps every safe input to a binary vector. This implication is what finite precision breaks: at the trigger, $\widehat{\text{STEP}}_\varepsilon(t_F) = 2$ while $\widehat{\text{RECT}}_\varepsilon(t_F) = 0$.

For example, in **float16** ($f = 10$), the trigger is $t_F = 513$. The exact values 512.75 and 512.25 are not representable and round to 513 and 512, so $\widehat{\text{STEP}}_{1/4}(513) = 2$ and $\widehat{\text{RECT}}_{1/4}(513) = 0$.

The same construction applies to the other formats considered in our evaluation, with the per-format triggers listed in Table 2. When the format is clear from context, we write t for t_F .

4.3 Key-Dependent Divergence and Key Recovery

The attack uses one chosen query for each key-bit position $i \in \{0, \dots, 127\}$. To target position i , the attacker sets the i -th input coordinate to t_F and every other coordinate to 0. By the analysis of the previous subsection, the finite-precision input processing maps the selected coordinate to 2 and every other coordinate to 0. The aggregate mask, the sum of the per-coordinate RECT values, is zero as well, since $\text{RECT}(t_F) = 0$ at the trigger coordinate and $\text{RECT}(0) = 0$ elsewhere.

The first keyed operation in the natural AES implementation is NN_{ARK} , which applies NN_{XOR} coordinate-wise between the transformed input and the first round key K_0 . By the coordinate-wise definition of NN_{ARK} recalled in Section 2.1, the selected coordinate becomes

$$\text{NN}_{\text{XOR}}(2, K_0[i]) = \begin{cases} 2, & K_0[i] = 0, \\ 1, & K_0[i] = 1. \end{cases}$$

For every other coordinate $j \neq i$, the input to NN_{ARK} is 0, and hence its output is

$$\text{NN}_{\text{XOR}}(0, K_0[j]) = K_0[j] \in \{0, 1\}.$$

The attacked key bit therefore determines whether the state after NN_{ARK} remains in the binary domain. If $K_0[i] = 1$, the selected coordinate is mapped from 2 to 1, while every other coordinate is already binary. The complete state after NN_{ARK} is therefore binary and coincides with the post- NN_{ARK} state obtained from the all-zero binary input. The subsequent natural AES computation follows its intended binary execution and produces a valid binary ciphertext.

If $K_0[i] = 0$, the selected coordinate instead remains 2. A non-binary value therefore enters the subsequent NN_{SB} operation. The natural S-box and Mix-Columns constructions are guaranteed to implement the corresponding AES operations correctly only on binary inputs, and the value 2 lies outside that domain. The resulting corrupted branch is propagated and amplified through the remaining natural AES operations, producing an output that is observably different from the binary branch.

In every format, the $K_0[i] = 1$ branch is a finite binary ciphertext, which the output-side STEP leaves unchanged. The representation of the corrupted branch depends on the format. In `bfloat16`, `float16`, and `float32`, the corrupted natural AES output overflows to a non-finite value (NaN or infinity). The output-side STEP maps NaN and $+\infty$ to non-finite outputs, while it maps $-\infty$ to 0. In `float64`, the corrupted natural AES output remains finite but has a very large magnitude, and the output-side STEP maps every coordinate to 0. At that magnitude, the shifts ε and $1 - \varepsilon$ of STEP fall far below the spacing $\Delta_F(e)$ of representable values, so $\text{fl}(x - \varepsilon) = \text{fl}(x - (1 - \varepsilon)) = x$ and $\widehat{\text{STEP}}(x) = 0$. This is the behavior we observe in every `float64` run. In each evaluated format, the corrupted ($K_0[i] = 0$) output is therefore degenerate (non-finite or of Hamming weight zero) and separable from the valid $K_0[i] = 1$ branch.

Consequently, the response to the query targeting coordinate i reveals the corresponding key bit:

$$K_0[i] = \begin{cases} 1, & \text{if the response follows the valid binary branch,} \\ 0, & \text{if the response exhibits the corrupted-branch signature.} \end{cases}$$

Repeating the query for all 128 coordinates recovers K_0 with 128 chosen queries.

Other values of ε . The mechanism is not specific to $\varepsilon = 1/4$. Whenever a representable input t has $\widehat{\text{RECT}}_\varepsilon(t) = 0$ and $1 < \widehat{\text{STEP}}_\varepsilon(t) \leq 2$, the attacked coordinate after NN_{ARK} becomes $|\widehat{\text{STEP}}_\varepsilon(t) - K_0[i]|$, which for $K_0[i] = 1$ lies in $[0, 1]$ and for $K_0[i] = 0$ equals $\widehat{\text{STEP}}_\varepsilon(t) > 1$, outside $[0, 1]$.

For $K_0[i] = 1$, the coordinate stays in $[0, 1]$ but is non-binary, so unlike the exact $\varepsilon = 1/4$ case (where that branch is exactly 1) the natural network's binary-input correctness no longer covers it. Whether the two branches are reliably distinguishable is therefore an empirical question. They are reliably distinguished only when $\widehat{\text{STEP}}_\varepsilon(t)$ is far enough above 1 to cause the corruption, as the experiments of Section 4.4 at $\varepsilon \in \{1/3, 1/5, 2/5\}$ show.

In Algorithm 2, $\text{round}(\cdot)$ is the coordinate-wise rounding of Section 3.2 and $\text{HW}(\cdot)$ is the Hamming weight of the result. An output is accepted as a valid

Algorithm 2: Floating-point key recovery on the protected implemen-
tation

Input: Protected oracle D_S in a format with f fraction bits; bound $\beta = 1.5$;
Hamming-weight threshold $\tau_H = 4$

Output: First round key K_0

$t \leftarrow 2^{f-1} + 1$

for $i \leftarrow 0$ **to** 127 **do**

$q \leftarrow \mathbf{0} \in F^{128}$; $q[i] \leftarrow t$

$out \leftarrow D_S(q)$

$valid \leftarrow (\text{all of } out \text{ finite}) \wedge (\max_j |out[j]| \leq \beta) \wedge (\text{HW}(\text{round}(out)) \geq \tau_H)$

$K_0[i] \leftarrow 1$ **if** $valid$ **else** 0

return K_0

ciphertext when all its coordinates are finite and at most $\beta = 1.5$, and its Hamming weight is at least $\tau_H = 4$. A valid ciphertext is binary with $\text{HW} \approx 64$, whereas a corrupted output is non-finite, exceeds 1, or has $\text{HW} \approx 0$. The values $\beta = 1.5$ and $\tau_H = 4$ therefore separate the two cases with a wide margin. The threshold τ_H is unrelated to the collision threshold τ of Section 3. This classifier is empirical, and Section 4.4 evaluates it.

4.4 Experimental Validation

The floating-point experiments use the same unmodified reference code as Section 3.3, again queried only as an oracle. Real values are evaluated in the network’s native floating-point format under PyTorch’s default numerics [17], and CPU runs pin the device so the behavior is deterministic.

In every format, the classifier of Algorithm 2 separates the two branches as predicted in the previous subsection. The numbers reported here are on the oracle D_S with the output-side STEP, our main target of Section 2.2. The released code of G  rault et al. [7], which applies STEP to the input only, gives identical recovery. The only difference is that its corrupted outputs in `float64` are large finite values rather than all zeros. Table 3 reports recovery over 100 random keys per format on each of the two networks. At the released $\varepsilon = 1/4$, the attack recovers the full first round key in every one of the 800 trials. A larger run of 10,000 random keys per format and network on the same oracle recovers all 80,000 (Appendix D).

The released code hard-codes $\varepsilon = 1/4$. For other values of ε , we implement STEP_ε and RECT_ε ourselves from the definitions of Section 2.2 and wrap them around the unmodified natural network D . Table 3 uses this implementation for every row, including $\varepsilon = 1/4$. The trigger depends on ε and is located by a short search over representable values. It is $2^{f-1} + 1$ at $\varepsilon = 1/5$, $2^{f-2} + 1$ at $\varepsilon = 1/3$, and $2^{f-4} + 1$ at $\varepsilon = 2/5$.

Below we quote $\text{STEP}(t)$ as computed with exact ε . The value obtained in a given format differs slightly, because ε itself is rounded to the working format. At

Table 3. First round key recovery on the protected construction across ε . Each entry is the number of keys, out of 100 random keys, whose first round key is fully recovered, per format, for the base network (Base) and the T-tables network (T-tables). Recovery is complete except as $\text{STEP}(t) \rightarrow 1$, at $\varepsilon = 2/5$. The $\text{STEP}(t)$ column lists the values computed with exact ε .

ε	$\text{STEP}(t)$	Network	<code>bfloat16</code>	<code>float16</code>	<code>float32</code>	<code>float64</code>	Total
1/4	2	Base	100	100	100	100	400
		T-tables	100	100	100	100	400
1/3	3/2	Base	100	100	100	100	400
		T-tables	100	100	100	100	400
1/5	5/3	Base	100	100	100	100	400
		T-tables	100	100	100	100	400
2/5	5/4	Base	92	80	72	82	326
		T-tables	83	80	82	85	330

$\varepsilon = 1/3$ and $\varepsilon = 1/5$ ($\text{STEP}(t) = 3/2$ and $5/3$), recovery is complete in all 800 trials each. Recovery falls off only as $\text{STEP}(t) \rightarrow 1$, where the corrupted ($K_0[i] = 0$) value $\text{STEP}(t)$ marginally leaves the valid range and the detection margin shrinks. At $\varepsilon = 2/5$ ($\text{STEP}(t) = 5/4$), 144 of the 800 trials fail. Table 3 gives the per-format, per-network counts. Probing each bit under several plaintexts instead of one removes these failures in `float16`, `float32`, and `float64`, but not in `bfloat16` (Appendix C).

5 A Defense Valid in Finite Precision

Section 4 shows that the defense of G  rault et al., though provably secure over the reals, has a gap in finite precision. The gap lies in the region of large values, where the representable spacing widens and precision is lowest. There STEP is no longer evaluated as its exact-real definition prescribes, and a single out-of-range coordinate slips past the sanitizer.

The fix is to force every input into $[0, 1]$ before it reaches the input-side STEP , so that the sanitizer is evaluated only on the bounded range $[0, 1]$, where the large-magnitude rounding failure of Section 4 cannot occur. We first show that prepending the ClippedReLU of G  rault et al. blocks our attack (Section 5.1). We then show that its subtraction-based realization does not guarantee outputs in $[0, 1]$ in finite precision, and give a rounding-robust realization that provably confines every finite representable input to $[0, 1]$ with the same ReLU count (Section 5.2).

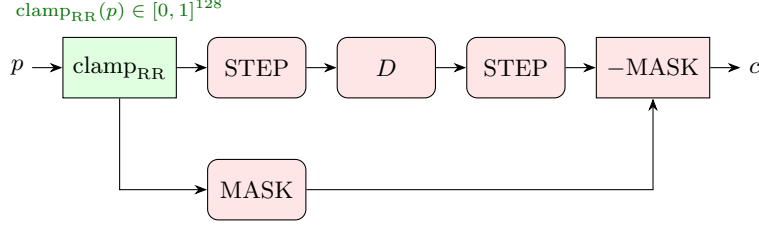


Fig. 3. The protected natural AES pipeline (red) and our saturating clamp clamp_{RR} (green). Prepending clamp_{RR} maps every finite input into $[0, 1]$ before it reaches STEP and MASK, so the sanitizer is only ever evaluated on the range where its exact-real definition holds. Over the reals, clamp_{CR} of Section 5.1 realizes the same clamp. The two agree as real functions and differ only in finite precision, where clamp_{RR} keeps every finite input in $[0, 1]$ while clamp_{CR} can leave a large input outside it (Section 5.2).

5.1 ClippedReLU Clamp

Gérault et al. [7] already introduce a saturating clamp, called ClippedReLU, which we denote by

$$\text{clamp}_{\text{CR}}(x) = \text{ReLU}(x) - \text{ReLU}(x - 1).$$

Over the reals, $\text{clamp}_{\text{CR}}(x) = \min(\max(x, 0), 1)$. Prepending clamp_{CR} immediately before the input-side STEP, where Figure 3 places clamp_{RR} , blocks the attack of Section 4. For the trigger t , $\text{clamp}_{\text{CR}}(t) = 1$ in the working precision, so the following STEP returns 1 and the coordinate is binarized as intended. Moreover, clamp_{CR} changes neither STEP nor RECT and leaves the exact-real Perfect Simulation intact.

This realization, however, carries a latent hazard. It is a clamp to $[0, 1]$ for real-valued inputs, yet in finite precision it is not. For a large input x , $\text{ReLU}(x) - \text{ReLU}(x - 1)$ may fall outside $[0, 1]$ after rounding. In `float16`, for instance, at $x = 2050$ the value $x - 1 = 2049$ rounds to 2048, so $\text{clamp}_{\text{CR}}(2050) = 2050 - 2048 = 2 \notin [0, 1]$. The attack of Section 4 is still blocked, because STEP maps this 2 back to 1. However, that is a coincidence of STEP, not a range guarantee of clamp_{CR} itself.

5.2 Rounding-Robust Clamp

We realize the same clamp instead as

$$\text{clamp}_{\text{RR}}(x) = 1 - \text{ReLU}(1 - \text{ReLU}(x)),$$

which uses the same two ReLU units as clamp_{CR} . Lemma 1 shows that clamp_{RR} keeps every finite representable input in $[0, 1]$ in the working precision, leaves binary inputs unchanged, and agrees with clamp_{CR} over the reals. This blocks every trigger that relies on an out-of-range value reaching STEP, not only the one of Section 4.

Lemma 1 (Bounded and exact-real transparent). *Under round-to-nearest arithmetic, every finite representable input is mapped by clamp_{RR} into $[0, 1]$, with $\text{clamp}_{\text{RR}}(x) = 0$ for $x \leq 0$ and $\text{clamp}_{\text{RR}}(x) = 1$ for $x \geq 1$. In particular, every per-format trigger maps to 1. Over exact real arithmetic,*

$$\text{STEP}(\text{clamp}_{\text{RR}}(x)) = \text{STEP}(x), \quad \text{RECT}(\text{clamp}_{\text{RR}}(x)) = \text{RECT}(x)$$

for all real x .

Proof. We evaluate clamp_{RR} in the representable set F , using three properties of rounding to nearest (fl , Section 4.1) that hold in every IEEE 754 binary format [9] and in `bf16` [11]:

- (i) $\text{fl}(z) = z$ for every $z \in F$, and $0, 1 \in F$.
- (ii) fl is monotone: $y \leq z \Rightarrow \text{fl}(y) \leq \text{fl}(z)$.
- (iii) For a representable input $y \in F$, $\text{ReLU}(y) = \max(y, 0)$ is computed exactly, since its value is y or 0, both in F .

From (i)–(ii), for every real y and every $z \in F$,

$$y \leq z \Rightarrow \text{fl}(y) \leq z, \quad y \geq z \Rightarrow \text{fl}(y) \geq z.$$

For a finite real input x , the evaluation first rounds it to $\text{fl}(x) \in F$. Since fl is monotone with $\text{fl}(0) = 0$ and $\text{fl}(1) = 1$, the cases $x \leq 0$ and $x \geq 1$ carry over to $\text{fl}(x)$. It therefore suffices to prove the claim for a representable argument, which we again denote $x \in F$. We evaluate $\text{clamp}_{\text{RR}}(x)$ in four steps:

$$a = \text{ReLU}(x) \rightarrow b = \text{fl}(1 - a) \rightarrow c = \text{ReLU}(b) \rightarrow d = \text{fl}(1 - c),$$

so that $\text{clamp}_{\text{RR}}(x) = d$. The two ReLU inputs x and $b = \text{fl}(1 - a)$ lie in F (x by assumption, b as a rounding output), so both ReLUs are exact by (iii), and only the two subtractions round.

We first show that the output always lands in $[0, 1]$. As $a = \max(x, 0) \geq 0$, we have $1 - a \leq 1$. By monotonicity of rounding, $b \leq 1$, and hence $c = \max(b, 0) \leq 1$. Therefore $1 - c \geq 0$, and monotonicity of rounding again gives $d \geq 0$. Since also $c \geq 0$, we have $1 - c \leq 1$ and therefore $d \leq 1$. Thus $\text{clamp}_{\text{RR}}(x) \in [0, 1]$ for every finite x , with no hypothesis on the precision.

We next evaluate the two endpoints, showing $\text{clamp}_{\text{RR}}(x) = 0$ for $x \leq 0$ and $\text{clamp}_{\text{RR}}(x) = 1$ for $x \geq 1$. If $x \leq 0$, then $a = 0$, $b = \text{fl}(1) = 1$, $c = 1$, and $d = \text{fl}(0) = 0$. If $x \geq 1$, then $a = x$ and $1 - a \leq 0$. By monotonicity of rounding, $b \leq 0$, so $c = \max(b, 0) = 0$ and $d = \text{fl}(1) = 1$. In particular, a trigger $t > 1$ gives $\text{clamp}_{\text{RR}}(t) = 1$ exactly in every format, supplying the high-side saturation that STEP fails to deliver.

Finally, we show that over exact real arithmetic the clamp changes neither STEP nor RECT. Here the ReLUs and subtractions are exact, so $\text{clamp}_{\text{RR}}(x) = \min(\max(x, 0), 1)$. From the definition of STEP, it equals 0 on $(-\infty, \varepsilon]$ and 1 on $[1 - \varepsilon, \infty)$, while RECT is 0 on $(-\infty, 0] \cup [1, \infty)$. For $x \in [0, 1]$, $\text{clamp}_{\text{RR}}(x) = x$. For $x < 0$, $\text{clamp}_{\text{RR}}(x) = 0$ with $\text{STEP}(x) = \text{STEP}(0) = 0$ and $\text{RECT}(x) = \text{RECT}(0) = 0$. For $x > 1$, $\text{clamp}_{\text{RR}}(x) = 1$ with $\text{STEP}(x) = \text{STEP}(1) = 1$ and $\text{RECT}(x) = \text{RECT}(1) = 0$. Hence STEP and RECT agree at x and at $\text{clamp}_{\text{RR}}(x)$ for every real x . $\square$

Cost and limitations. clamp_{RR} uses two ReLU units per input coordinate, exactly as clamp_{CR} does, so the rounding-robust realization adds no ReLU unit beyond the clamp of G  rault et al., only one layer of depth. This is an attack-specific hardening result, not a general finite-precision security theorem. The clamp acts only at the input boundary, and none of the surrounding STEP and MASK mechanisms reaches the internal rounds of D . A finite-precision trigger that first arises inside D would therefore escape all three. Whether such an internal-round trigger exists is left open.

Experimental validation. We validate the defense by prepending clamp_{RR} to the unmodified reference code. Over every finite value of `bfloat16`, `float16`, and `float32`, clamp_{RR} stays in $[0, 1]$, whereas clamp_{CR} leaves it on 64, 512, and 4,194,304 inputs respectively, always with the value 2. In `float64`, 2×10^7 random finite values leave clamp_{RR} in $[0, 1]$, while clamp_{CR} leaves it on 2,455 of them. Against the clamped network, the attack of Section 4 learns nothing, since every query returns a valid ciphertext. Binary plaintexts pass through clamp_{RR} unchanged, and the clamped network reproduces the FIPS-197 test vector.

6 Conclusion

We revisited the natural DNN implementation of AES of G  rault et al. and its provably secure defense. On the unprotected implementation, we extended the absorption property of the natural S-box one round deeper than the known attacks. We proposed a chosen-plaintext collision test that recovers the second round key column by column, and with it the full AES-256 master key in about 2^{39} queries. On the protected implementation, the Perfect Simulation theorem holds over the reals but does not survive finite precision. We found an input that escapes the sanitizer in every binary floating-point format we tested, and 128 chosen queries recover the first round key. Finally, we proposed a rounding-robust realization of the saturating clamp that closes this channel while leaving the exact-real construction unchanged.

We experimentally validated the attacks and defense, and all code and logs behind the numbers reported in this paper are available at <https://github.com/cqc-lab/breaking-deep-neural-cryptography>. Appendix D maps each result to the script that produced it and to its committed artifact.

Use of AI tools. We used Claude (Anthropic, Claude Fable 5.1, through Claude Code) and ChatGPT (OpenAI, GPT-5.6 Sol) as assistants in preparing this work. These tools assisted with drafting and revising text throughout this paper, writing and debugging parts of the artifact code, and checking numerical claims against our experimental logs. The attacks, analysis, defense, and experiments were designed and carried out by us. We reviewed every generated sentence and line of code and take full responsibility for the correctness of the work.

References

1. Bao, Z., Lu, J., Yao, Y., Zhang, L.: More insight on deep learning-aided cryptanalysis. In: *Advances in Cryptology – ASIACRYPT 2023*. LNCS, vol. 14440, pp. 436–467. Springer (2023). https://doi.org/10.1007/978-981-99-8727-6_15
2. Benamira, A., G  rault, D., Peyrin, T., Tan, Q.Q.: A deeper look at machine learning-based cryptanalysis. In: *Advances in Cryptology – EUROCRYPT 2021*. LNCS, vol. 12696, pp. 805–835. Springer (2021). https://doi.org/10.1007/978-3-030-77870-5_28
3. Canales-Mart  nez, I.A., Ch  vez-Saab, J., Hambitzer, A., Rodr  guez-Henr  quez, F., Satpute, N., Shamir, A.: Polynomial time cryptanalytic extraction of neural network models. In: *Advances in Cryptology – EUROCRYPT 2024*. LNCS, vol. 14653, pp. 3–33. Springer (2024). https://doi.org/10.1007/978-3-031-58734-4_1
4. Carlini, N., Jagielski, M., Mironov, I.: Cryptanalytic extraction of neural network models. In: *Advances in Cryptology – CRYPTO 2020*. LNCS, vol. 12172, pp. 189–218. Springer (2020)
5. Chen, Y., Yu, H.: Bridging machine learning and cryptanalysis via EDLCT. *Cryptology ePrint Archive*, Paper 2021/705 (2021)
6. Daemen, J., Rijmen, V.: *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer (2002)
7. G  rault, D., Hambitzer, A., Ronen, E., Shamir, A.: Deep neural cryptography. In: *Advances in Cryptology – EUROCRYPT 2026*. LNCS, vol. 16546, pp. 513–543. Springer (2026). https://doi.org/10.1007/978-3-032-25333-0_18
8. Gohr, A.: Improving attacks on round-reduced Speck32/64 using deep learning. In: *Advances in Cryptology – CRYPTO 2019*. LNCS, vol. 11693, pp. 150–179. Springer (2019)
9. IEEE: IEEE standard for floating-point arithmetic (IEEE 754-2019) (2019). <https://doi.org/10.1109/IEEESTD.2019.8766229>
10. Jin, J., McMurtry, E., Rubinstein, B.I.P., Ohrimenko, O.: Are we there yet? Timing and floating-point attacks on differential privacy systems. In: *IEEE Symposium on Security and Privacy (S&P)*. pp. 473–488 (2022)
11. Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., et al.: A study of BFLOAT16 for deep learning training. *arXiv preprint arXiv:1905.12322* (2019)
12. Klimov, A., Mityagin, A., Shamir, A.: Analysis of neural cryptography. In: *Advances in Cryptology – ASIACRYPT 2002*. LNCS, vol. 2501, pp. 288–298. Springer (2002)
13. Lin, X., Tibouchi, M., Yu, Y., Zhang, S.: Do not disturb a sleeping falcon: Floating-point error sensitivity of the falcon sampler and its consequences. In: *Advances in Cryptology – EUROCRYPT 2025*. LNCS, vol. 15602, pp. 213–244. Springer (2025). https://doi.org/10.1007/978-3-031-91124-8_8
14. Mironov, I.: On significance of the least significant bits for differential privacy. In: *ACM Conference on Computer and Communications Security (CCS)*. pp. 650–661 (2012)
15. National Institute of Standards and Technology: FIPS 197: Advanced encryption standard (AES) (2001), updated 2023
16. Pan, X., Srikumar, V.: Expressiveness of rectifier networks. In: *International Conference on Machine Learning (ICML)*. PMLR, vol. 48, pp. 2427–2435 (2016)
17. PyTorch Contributors: Numerical accuracy. PyTorch documentation, https://pytorch.org/docs/stable/notes/numerical_accuracy.html (2024)

18. Yuan, Y., Wu, W., Zhang, L., Wu, R.: Fourier analysis of neural distinguishers. *IACR Transactions on Symmetric Cryptology (ToSC)* **2026**(1) (2026)
19. Zhang, L., Yao, Y., Shi, D., Chai, D., Guo, J., Wang, Z.: Neural-inspired advances in integral cryptanalysis. In: *Advances in Cryptology – EUROCRYPT 2026*. LNCS, vol. 16546, pp. 451–481. Springer (2026)

Supplementary Material

A Analysis of the Collision Test

This appendix establishes the absorption property used in Section 3, follows a probe through round 1 to the four round-2 S-boxes, and derives the scores of the correct and of the wrong candidates over the reals, which proves Proposition 1.

A.1 Absorption at 0x52

Recall that the corner function for $b \in \{0, 1\}^8$ is

$$\text{corner}_b(x) = \text{ReLU}\left(\sum_{i:b_i=1} x_i + \sum_{i:b_i=0} (1 - x_i) - 8 + 1\right),$$

and that the natural S-box is $\text{NN}_{\text{SB}}(x) = \sum_b \text{corner}_b(x) \cdot \text{SBox}(b)$. Lemma 2 gives the output of the natural S-box when a binary input is perturbed inward, and shows that only 0x52 can absorb the perturbation.

Lemma 2 (Absorption criterion). *Write e_j for the unit vector at coordinate j . Take a Boolean input $v \in \{0, 1\}^8$ and perturb each coordinate i inward by $\eta_i \geq 0$ (a 1-bit becomes $1 - \eta_i$, a 0-bit becomes η_i). Write $\tilde{v}$ for the result and $S = \sum_i \eta_i$. If $0 < S < 1$, then*

$$\text{NN}_{\text{SB}}(\tilde{v}) = (1 - S) \text{SBox}(v) + \sum_{j: \eta_j > S/2} (2\eta_j - S) \text{SBox}(v \oplus e_j).$$

In particular, at $v = 0\text{x}52$ the output stays 0 if and only if $\eta_j \leq S/2$ for every j . At any $v \neq 0\text{x}52$, the output changes.

Proof. For a corner b , let $\Delta = \{i : b_i \neq v_i\}$ and $d = |\Delta|$. In $\tilde{v}$, a coordinate that agrees with b contributes $1 - \eta_i$ and one that disagrees contributes η_i , so the argument of the ReLU in $\text{corner}_b(\tilde{v})$ is

$$\sum_{i \notin \Delta} (1 - \eta_i) + \sum_{i \in \Delta} \eta_i - 7 = 1 - d - S + 2 \sum_{i \in \Delta} \eta_i,$$

using $\sum_{i \notin \Delta} \eta_i = S - \sum_{i \in \Delta} \eta_i$. Since $\sum_{i \in \Delta} \eta_i \leq S < 1$, for $d \geq 2$ this argument is at most $1 - d + S \leq S - 1 < 0$, so every corner with $d \geq 2$ is inactive. The corner $b = v$ ($d = 0$) has value $1 - S > 0$. A corner $b = v \oplus e_j$ ($d = 1$) has value $\text{ReLU}(2\eta_j - S)$, which is positive if and only if $\eta_j > S/2$. Summing the active corners gives the stated formula. At most one index j satisfies $\eta_j > S/2$.

At $v = 0\text{x}52$, $(1 - S)\text{SBox}(v) = 0$. If some j satisfies $\eta_j > S/2$, the output is $(2\eta_j - S)\text{SBox}(v \oplus e_j)$, which is nonzero because the S-box is a bijection. Otherwise the output is 0.

At $v \neq 0\text{x}52$, $\text{SBox}(v) \neq 0$. If no j satisfies $\eta_j > S/2$, the output is $(1 - S)\text{SBox}(v) \neq \text{SBox}(v)$ because $S > 0$. If one does, the output equals $\text{SBox}(v)$ only if $S\text{SBox}(v) = (2\eta_j - S)\text{SBox}(v \oplus e_j)$. This is impossible, since $\text{SBox}(v)$ and $\text{SBox}(v \oplus e_j)$ differ in at least one bit. $\square$

The attack delivers perturbations of one common magnitude on several coordinates, and Corollary 1 covers that case.

Corollary 1. *Let v , η_i , and $\tilde{v}$ be as in Lemma 2, without the assumption $S < 1$, and suppose that exactly n of the η_i equal η and the rest are 0. If $n = 1$ and $0 < \eta < \frac{1}{2}$, then $\text{NN}_{\text{SB}}(\tilde{v}) = (1 - \eta)\text{SBox}(v) + \eta\text{SBox}(v \oplus e_j)$, where j is the perturbed coordinate. If $n \geq 2$ and $0 < \eta < \frac{1}{2}$, then $\text{NN}_{\text{SB}}(\tilde{v}) = \text{ReLU}(1 - n\eta)\text{SBox}(v)$, even when $n\eta \geq 1$. In particular, among the perturbed inputs ($n \geq 1$), the output at `0x52` is unchanged if and only if $n \geq 2$, and at any $v \neq \text{0x52}$ the output changes for every $n \geq 1$.*

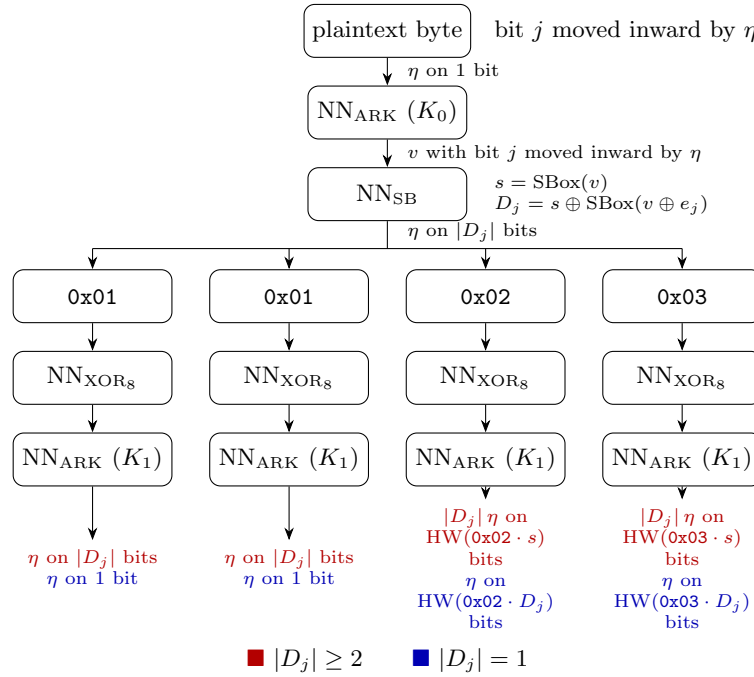
Proof. Let $J = \{i : \eta_i = \eta\}$. The corner argument computed in the proof of Lemma 2 equals $1 - d - n\eta + 2\eta|\Delta \cap J|$. For $n = 1$, this is $1 - \eta$ at $d = 0$, η at $\Delta = J$, and negative otherwise, which gives the first formula. For $n \geq 2$ and $d \leq n$, it is at most $1 - n\eta - d(1 - 2\eta)$, which is at most $\eta(2 - n) \leq 0$ for $d \geq 1$ since $\eta < \frac{1}{2}$. For $d > n$, it is at most $n(\eta - 1) < 0$. Only $d = 0$ can be active, with value $\text{ReLU}(1 - n\eta)$. $\square$

Lemma 2 and Corollary 1 use only the corner structure of NN_{SB} . They therefore hold unchanged for the natural implementation of any function $\{0, 1\}^\kappa \rightarrow \{0, 1\}^m$, with 8 replaced by κ in the corner function, SBox replaced by that function, and `0x52` by any input that it maps to 0. Appendices A.2 and A.3 apply them to the `0x02` and `0x03` blocks and to NN_{XOR_8} of the base network, and to the combined S-box and coefficient blocks of the T-tables network.

A.2 Propagation through Round 1

We now follow a probe through round 1. Suppose the probe perturbs bit j of the byte v that enters a round-1 S-box. The first NN_{ARK} preserves the perturbation, so this S-box receives v with coordinate j moved inward by η . Write $s = \text{SBox}(v)$ for its binary output. By Corollary 1 for a single perturbed bit, the S-box outputs s with every bit of $D_j = s \oplus \text{SBox}(v \oplus e_j)$ moved inward by η . We abbreviate $|D_j| = \text{HW}(D_j)$, the Hamming weight of D_j . Figure 4 summarizes the path of the probe through round 1.

In MixColumns, this byte is multiplied by `0x01`, `0x01`, `0x02`, and `0x03` and XORed into the four bytes of the target column. Multiplication by `0x01` leaves it unchanged, so those two branches carry η on $|D_j|$ bits. The `0x02` and `0x03` blocks are themselves natural blocks, and which branch of Corollary 1 governs them depends on $|D_j|$. Both NN_{XOR_8} and the second NN_{ARK} preserve an inward perturbation. Since the probe perturbs a single source byte, NN_{XOR_8} sees at most one non-binary input per bit position and returns the binary parity with that bit moved inward by the same amount, and NN_{ARK} computes $|x - K|$ against a binary key bit. Provided $\eta < \frac{1}{16}$, every magnitude that arrives is below $\frac{1}{2}$, which is the hypothesis of Corollary 1. A perturbed target byte is therefore absorbed if and only if it equals `0x52` and the perturbation occupies at least two of its bits, and a target byte left unperturbed is trivially absorbed whatever its value. A probe collides when all four targets are absorbed. The non-target columns of



A probe collides if all four target bytes are absorbed.

$|D_j| \geq 2$: The **0x01** targets are absorbed if they equal **0x52**.

The **0x02** and **0x03** targets are absorbed if $s = 0x00$,
or if they equal **0x52** and receive at least two bits.

$|D_j| = 1$: The **0x01** targets are never absorbed, so the probe never collides.

Fig. 4. Path of a probe through round 1 on the base network.

M_1 are binary and unperturbed throughout, so whether a probe is absorbed, and hence whether it collides by absorption, does not depend on them.

Case $|D_j| \geq 2$. Corollary 1 with $n = |D_j|$ gives $\text{ReLU}(1 - |D_j|\eta) \cdot (0x02 \cdot s)$, the binary product with each of its *one*-bits moved inward by $|D_j|\eta$ and its zero bits left at 0, and likewise for **0x03**. Each target S-box therefore receives a perturbation of a single magnitude, namely

coefficient **0x01** : η on $|D_j|$ bits,
 coefficient **0x02** : $|D_j|\eta$ on $\text{HW}(0x02 \cdot s)$ bits,
 coefficient **0x03** : $|D_j|\eta$ on $\text{HW}(0x03 \cdot s)$ bits.

The last two counts are 0 exactly when $s = 0x00$. Both products then vanish and the two branches carry nothing at all. At the correct candidate, all four

target bytes sit at 0x52, and a probe collides by absorption if and only if

$$\begin{cases} \min(|D_j|, \text{HW}(0x02 \cdot s), \text{HW}(0x03 \cdot s)) \geq 2, & s \neq 0x00, \\ |D_j| \geq 2, & s = 0x00, \end{cases}$$

the second line because the 0x02 and 0x03 targets are then absorbed for free.

Case $|D_j| = 1$. The single-bit case of Corollary 1 applies to the two blocks instead. They output $(1 - \eta) 0x02 \cdot s + \eta 0x02 \cdot (s \oplus D_j)$, which by linearity of the field multiplication is $0x02 \cdot s$ with the bits of $0x02 \cdot D_j$ moved inward by η . They thus carry η on $\text{HW}(0x02 \cdot D_j)$ bits, and likewise for 0x03. The two coefficient-0x01 targets, however, are perturbed on a single bit and are not absorbed at any value, so such a probe never collides by absorption, whatever the other two branches do. The case $|D_j| = 1$ therefore never contributes a forced collision, at the correct candidate or at a wrong one.

A.3 Wrong Candidates that Collide

Absorption also forces collisions at *wrong* candidates, and these are not rare. Each source byte of the column reaches two target bytes through 0x02 and 0x03 and the other two through 0x01. We call these other two the 0x01 targets of that source. By Figure 4, a probe collides by absorption at a wrong candidate g exactly when $s = 0x00$ and the two 0x01 targets of its source equal 0x52. If $s \neq 0x00$, all four targets are perturbed and all four must equal 0x52, which happens only at g^* .

At $s = 0x00$, seven of the eight probes have $|D_j| \geq 2$, the exception being the flip to 0x72 with $\text{SBox}(0x72) = 0x40$ of weight 1, so such a source contributes 7 collisions. We now count the wrong candidates g that satisfy the two conditions for a given source. The candidate g has four bytes. The two 0x01 targets fix two of them. The condition $s = 0x00$ concerns the S-box output of the source byte, and the four S-box outputs are $\text{MC}^{-1}(g)$, a linear function of g . It is therefore one linear equation over $\text{GF}(2^8)$ in the two remaining bytes of g and leaves one of them free. This gives 256 candidates per source and 1024 for the four sources of a column.¹ A wrong candidate thus scores 7 for each of its sources with $s = 0x00$ whose 0x01 targets equal 0x52.

The 0x01 targets of two different sources cover three bytes of the target column when the sources sit in consecutive rows, cyclically, and all four bytes otherwise. Any two non-consecutive sources, and any three sources, therefore force every byte of the target column to 0x52 and hence $g = g^*$. A wrong candidate thus has one or two such sources, two only when they are consecutive, and scores 7 or 14 accordingly. The score 7 is the one observed in Section 3.3. The 0x01 targets of both sources of a consecutive pair fix three bytes of g , and

¹ A few of the 1024 can coincide, either with each other or with g^* . Over 200,000 random key columns, 96.9% have exactly 1024 distinct ones and the rest 1023 or 1022 (Appendix D).

$s = 0x00$ for each source then determines the fourth. For a uniformly random key column, the two determinations agree with probability exactly $1/256$, since their agreement is the one linear condition that the three fixed bytes of g^* XOR to $0x00$. The four conditions, one per consecutive pair, are linearly independent, so a column admits a candidate scoring 14 with probability $1 - (255/256)^4 \approx 1.553\%$. This proves Proposition 1(ii).

This analysis is specific to the base network, where $0x02$ and $0x03$ are natural blocks applied to the S-box output. In the T-tables network, they are combined with the S-box into one block (Section 2.1), so a probe enters that block on a single bit and, by Corollary 1, the $0x02$ and $0x03$ targets receive η on $\text{HW}(0x02 \cdot D_j)$ and $\text{HW}(0x03 \cdot D_j)$ bits even when $s = 0x00$. No target is then absorbed for free, a probe collides only if all four targets equal $0x52$, and the colliding wrong candidates disappear. The criterion for the correct candidate becomes $\min(|D_j|, \text{HW}(0x02 \cdot D_j), \text{HW}(0x03 \cdot D_j)) \geq 2$ for every source, under which its exact-real score is at least 20 (Appendix A.4).

A.4 Structure of the Correct Candidate's Score

We now prove Proposition 1(i). At the correct candidate, the four source S-box outputs $s_0, \dots, s_3$ are $\text{MC}^{-1}(g^*)$ with $g^* = K_{1,c} \oplus (0x52, 0x52, 0x52, 0x52)$. For a uniformly random key column, g^* is uniform, and MC^{-1} is a bijection of $\text{GF}(2^8)^4$, so $s_0, \dots, s_3$ are independent and uniform. By the criterion of Appendix A.2, a source with output s contributes

$$\begin{cases} \#\{j : |D_j| \geq 2\}, & s \neq 0x00, \text{HW}(0x02 \cdot s) \geq 2, \text{HW}(0x03 \cdot s) \geq 2, \\ 7, & s = 0x00, \\ 0, & \text{otherwise} \end{cases}$$

forced collisions. From the S-box table, $\#\{j : |D_j| \geq 2\} \in \{7, 8\}$, and since $0x02$ and $0x03$ are invertible, $\text{HW}(0x02 \cdot s) \leq 1$ or $\text{HW}(0x03 \cdot s) \leq 1$ holds for 17 values of s , among them $s = 0x00$. Hence each contribution is 0, 7, or 8, for 16, 46, and 194 values of s respectively, and N_{g^*} is the sum of four independent such contributions, with the distribution stated in Proposition 1(i). The event $N_{g^*} < 8$ requires all four contributions to be 0 or exactly one to be 7, so $\Pr[N_{g^*} < 8] = (16/256)^4 + 4 \cdot (46/256)(16/256)^3 = 25/131072$. The distribution has its mass at 14–16, 21–24, and 28–32, the three groups of peaks visible in Figure 2, and the predicted mass below $\tau = 8$ is 0.019% per column against the measured 0.025% (10 of 40,000). On the T-tables network, the contribution of a source is $\#\{j : \min(|D_j|, \text{HW}(0x02 \cdot D_j), \text{HW}(0x03 \cdot D_j)) \geq 2\}$, which equals 5, 6, 7, or 8 for 1, 13, 83, and 159 values of s respectively, so $N_{g^*} \geq 20$ and $\Pr[N_{g^*} < 8] = 0$.

The agreement is not exact. Measured scores also fall between the predicted groups on 0.84% of the columns, and the highest score group, 28 to 32, holds less mass than predicted. Both deviations move mass downward, since a few probes that the exact-real criterion counts as absorbed do not collide in floating-point arithmetic. In `float32`, nine of the 10 columns below 8 have exact-real score 7,

and one of them measures 6 because one predicted collision is lost to rounding. The tenth has exact-real score 8 and measures 7 for the same reason. In `float16`, on the same keys, the correct candidate’s score differs from `float32` by at most 2 on 98.6% of the columns, and nine of the ten low columns are shared.

B Parameter Choices for the Second Round Key Recovery

This appendix explains the two parameters of the collision test of Section 3.2, the perturbation $\eta = 0.05$ and the threshold $\tau = 8$, and describes the fallback used when a column’s correct candidate scores below the threshold.

Choice of η . A probe perturbs one plaintext bit by η . Round 1 rescales it by a factor of at most $|D_j| \leq 8$ (Appendix A.2), so the largest magnitude delivered to a round-2 S-box is 8η . Corollary 1 covers every target S-box as long as $8\eta < \frac{1}{2}$, that is, $\eta < \frac{1}{16}$. Within this range, the value $\eta = 0.05$ is a practical choice. It is large enough to survive the network’s `float16` rounding, which a much smaller η would not. A smaller η would also not raise the score of the few correct candidates that score low. Their low scores come from probes that perturb only one bit of a round-2 S-box input. By Corollary 1, such a perturbation is never absorbed for any η . The loss is therefore structural, as described in Appendix A.4.

Choice of τ . Figure 2 shows the collision score of the correct candidate over the 40,000 columns of Section 3.3. The scores concentrate at 22 to 32, with a median of 31, and 10 columns fall below 8. A wrong candidate scores 0, 7, or 14 (Appendix A.3), so every threshold from 8 to 14 has the same effect on wrong candidates, and so does every threshold above 14. Since correct scores reach down to 5, no threshold separates the two regimes completely. With $\tau = 8$, Proposition 1 predicts that a wrong candidate passes the threshold in about 1.6% of the columns and that the correct candidate falls below it in about 0.019%. We confirm both rates experimentally, measuring 1.60% and 0.025% in the censuses of Section 3.3. Choosing $\tau = 15$ would exclude every wrong candidate, but it would lose the correct candidate more often.

Fallback for low scores of the correct candidate. Since no threshold separates the correct candidate from every wrong one, a column may fail the threshold test or the re-encryption check of Algorithm 1. Such a column can still be recovered by recording its candidates with a nonzero score in decreasing order. By Proposition 1, at most 1024 wrong candidates have a nonzero score, so the list stays short. Taking one candidate per column assembles a full $K_0 \| K_1$, and it can be verified with a known plaintext–ciphertext pair. This adds local verification only, not further oracle queries.

C Improving Recovery at $\varepsilon = 2/5$ with Multiple Plaintexts

At the low-margin value $\varepsilon = 2/5$ ($\text{STEP}(t) = 5/4$ with exact ε) the single-plaintext attack of Section 4 recovers only part of the key (Table 3). A $K_0[i] = 0$ coordinate is now only $1/4$ above 1, so it does not always corrupt the computation. For some base plaintexts, the perturbation stays bounded and the output-side STEP snaps it back to a clean binary ciphertext, which the valid-ciphertext oracle cannot tell from the $K_0[i] = 1$ branch. Whether the perturbation stays bounded depends on the surrounding (binary) plaintext, not only on the key.

This motivates probing each bit with multiple plaintexts. We take 16 base plaintexts, the all-zero one and 15 random binary ones, and place the trigger at position i of each. Over these 16 plaintexts, we count how many yield a corrupted output and read $K_0[i] = 0$ once this count reaches a threshold.

For `float16`, `float32`, and `float64`, a threshold of 2 recovers every key. However, `bfloat16` has only 7 fraction bits, and the non-binary $K_0[i] = 1$ coordinate itself overflows to a non-finite value under almost every base plaintext. Additional plaintexts therefore cannot separate the two branches, and voting recovers at most 107/200 keys even at the best threshold. Table 4 summarizes the result.

Table 4. Keys fully recovered at $\varepsilon = 2/5$, out of 200 trials per format (100 random keys on each of the two networks, as in Table 3), with a single plaintext and with 16 plaintexts.

Format	single plaintext	16 plaintexts
<code>bfloat16</code>	175/200	107/200
<code>float16</code>	160/200	200/200
<code>float32</code>	154/200	200/200
<code>float64</code>	167/200	200/200

D Reproduction

The artifact is available at <https://github.com/cqc-lab/breaking-deep-neural-cryptography>. It is split into two self-contained directories, `part1/` and `part2/`, matching Part I and Part II of Table 5. Table 5 lists, for every result reported in this paper, the script that produced it and the committed log or directory under `part1/artifacts/` or `part2/artifacts/`. Each directory carries an unmodified copy of the released `nn_aes.py` and `utils.py` of G  rault et al. [7]. The only runtime configuration applied to them is setting the module dtype to the target format before the network is built. The `README` documents the environment, the seeds, the license, and the options of each script.

Table 5. Reported results, the scripts that produced them, and their committed artifacts.

Result	Script	Artifact
<i>Part I: collision attack for K_1 and the full AES-256 key</i>		
Full censuses, two keys (Section 3.3)	<code>run_attack.py</code> <code>--record-all</code>	<code>run_seed2026/</code> , <code>run_seed2027/</code>
Full score histograms, third key, all four columns (Section 3.3)	<code>run_attack.py</code> <code>--record-all, HIST=1</code>	<code>run_hist.2028/</code>
Margin census in <code>float32</code> and <code>float16</code> (Section 3.3, Figure 2)	<code>estimate_k1_margin.py</code>	<code>fp32/</code> , <code>fp16/</code>
Enumeration of the candidates scoring 14 (Section 3.3)	<code>measure_score14.py</code>	<code>measure_score14.log</code>
Enumeration over 200,000 columns (Appendix A.3)	<code>enumerate_forced_</code> <code>candidates.py</code>	<code>forced_candidates.log</code> , <code>forced_candidates_</code> <code>ttables.log</code>
Full census on the T-tables network, fourth key (Section 3.3)	<code>run_attack.py</code> <code>--record-all, HIST=1,</code> <code>MODEL=ttables</code>	<code>run_hist.2029_ttables/</code>
Base-network colliding candidates on the T-tables key (Section 3.3)	<code>check_census_key.py</code>	<code>check_census_key_</code> <code>seed2029.base.log</code> , <code>check_census_key_</code> <code>seed2029.ttables.log</code>
<i>Part II: floating-point attack and defense</i>		
Triggers and $\widehat{\text{STEP}}(t_F) = 2$ in the four formats (Table 2)	<code>step_fp_trigger_</code> <code>experiment.py</code>	<code>step_fp_trigger_</code> <code>experiment.log</code>
Recovery on the released code (Section 4.4)	<code>fp_key_recovery.py</code>	<code>fp_key_recovery.log</code>
80,000 keys with the output-side STEP (Section 4.4)	<code>fp_key_recovery_</code> <code>output_step.py</code>	<code>fp_output_step_</code> <code>10000keys.log</code>
Sweep over ε (Table 3)	<code>fp_epsilon_</code> <code>independence.py</code>	<code>fp_epsilon_</code> <code>independence.log</code>
Validation of <code>clamp_{RR}</code> (Section 5.2)	<code>counter_fp.py</code>	<code>counter_fp.log</code>
Multi-plaintext voting (Table 4)	<code>fp_eps25_vote.py</code>	<code>fp_eps25_vote.log</code>
<code>bfloat16</code> diagnosis (Appendix C)	<code>fp_eps25_why_</code> <code>corrupt.py</code>	<code>fp_eps25_why_corrupt.log</code>