Going Beyond Constant-Time Security

Engineering & Verifying Physical-Attack Defenses in Open-Source Cryptography

Jan Janásek & Łukasz Chmielewski
Masaryk University

October 8, 2026

About Me

Jan Janásek
Masaryk University, Brno

Position

Master's student, Masaryk University

Research

side-channel analysis · fault injection · protected open-source crypto

Libraries

github.com/sca-protected-rsa-library
github.com/sca-secure-library-sca25519

LinkedIn

GitHub

Outline

01 **Method**
the hardening cycle · the testing pipeline · ChipWhisperer

02 **Five levels**
timing & SPA · blinding · TVLA, the gate · table position & reuse · faults

03 **Integration**
what is new · the layered architecture in BearSSL · what it costs: +56 %

04 **Takeaways**
three lessons for developers

05 **General observations**
two protected libraries · what is missing · getting it used

Methodology: The Hardening Cycle

Target: BearSSL's RSA private-key path — CRT, windowed exponentiation — on Cortex-M4.

Five attack levels

1. **Timing & SPA** — branches and execution path

Methodology: The Hardening Cycle

Target: BearSSL's RSA private-key path — CRT, windowed exponentiation — on Cortex-M4.

Five attack levels

1. **Timing & SPA** — branches and execution path
2. **CPA** — the window-table entry

attack details in the appendix

Methodology: The Hardening Cycle

Target: BearSSL's RSA private-key path — CRT, windowed exponentiation — on Cortex-M4.

Five attack levels

1. **Timing & SPA** — branches and execution path
2. **CPA** — the window-table entry
3. **TVLA** — leakage detection

attack details in the appendix

Methodology: The Hardening Cycle

Target: BearSSL's RSA private-key path — CRT, windowed exponentiation — on Cortex-M4.

Five attack levels

1. **Timing & SPA** — branches and execution path
2. **CPA** — the window-table entry
3. **TVLA** — leakage detection
4. **Table position & reuse** — address-bit DPA, horizontal

attack details in the appendix

Methodology: The Hardening Cycle

Target: BearSSL's RSA private-key path — CRT, windowed exponentiation — on Cortex-M4.

Five attack levels

1. **Timing & SPA** — branches and execution path
2. **CPA** — the window-table entry
3. **TVLA** — leakage detection
4. **Table position & reuse** — address-bit DPA, horizontal
5. **Fault injection** — Bellcore CRT factorisation

attack details in the appendix

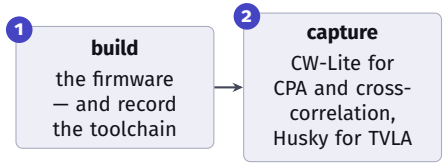
The Testing Pipeline: How Every Layer Is Earned

1

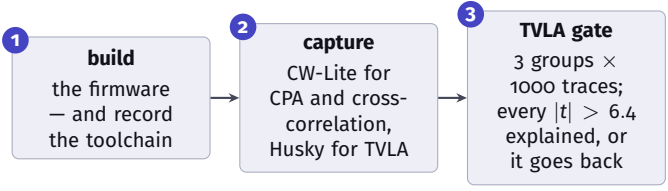
build

the firmware
— and record
the toolchain

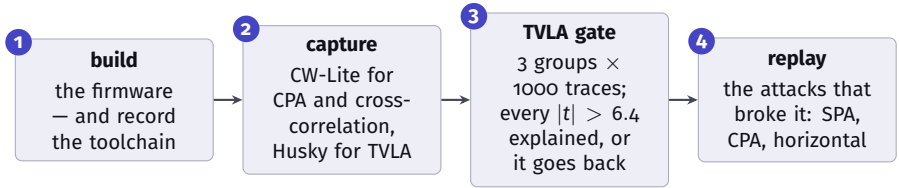
The Testing Pipeline: How Every Layer Is Earned



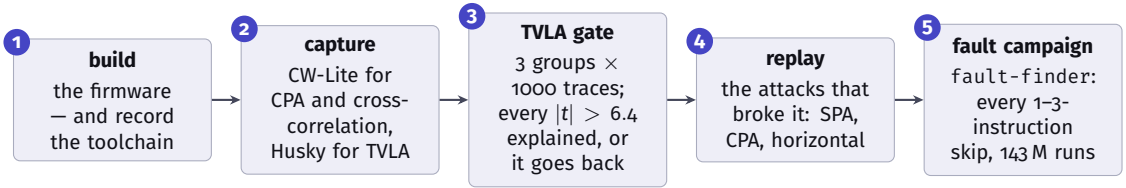
The Testing Pipeline: How Every Layer Is Earned



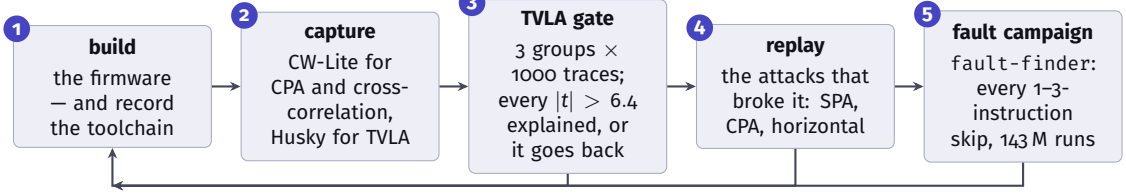
The Testing Pipeline: How Every Layer Is Earned



The Testing Pipeline: How Every Layer Is Earned

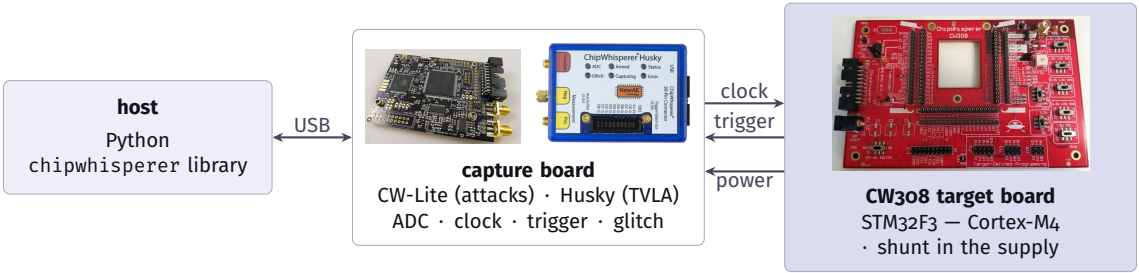


The Testing Pipeline: How Every Layer Is Earned



ChipWhisperer: The Bench Is a Python Script

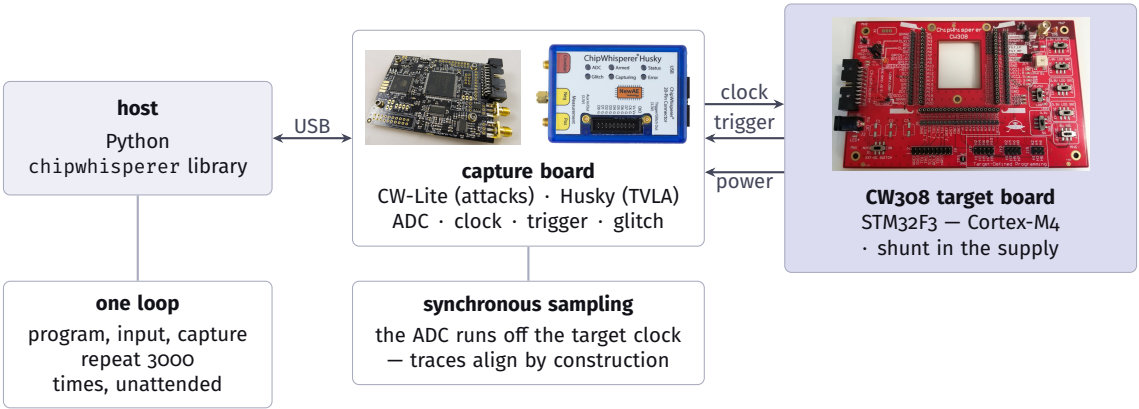
open-source capture platform (NewAE) — the same tool for the attacks and for the TVLA gate



photos: NewAE Technology — chipwhisperer.readthedocs.io

ChipWhisperer: The Bench Is a Python Script

open-source capture platform (NewAE) — the same tool for the attacks and for the TVLA gate



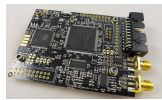
photos: NewAE Technology — chipwhisperer.readthedocs.io

ChipWhisperer: The Bench Is a Python Script

open-source capture platform (NewAE) — the same tool for the attacks and for the TVLA gate

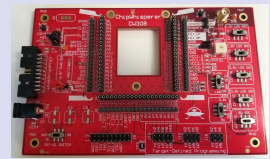
host
Python
`chipwhisperer` library

USB



capture board
CW-Lite (attacks) · Husky (TVLA)
ADC · clock · trigger · glitch

clock
trigger
power



CW308 target board
STM32F3 — Cortex-M4
· shunt in the supply

one loop
program, input, capture
repeat 3000
times, unattended

synchronous sampling
the ADC runs off the target clock
— traces align by construction

external scope for long traces
PicoScope 3406D · Riscure
current probe · BLP-50+
the TVLA runs

photos: NewAE Technology — chipwhisperer.readthedocs.io

Outline

01 **Method**
the hardening cycle · the testing pipeline · ChipWhisperer

02 **Five levels**
timing & SPA · blinding · TVLA, the gate · table position & reuse · faults

03 **Integration**
what is new · the layered architecture in BearSSL · what it costs: +56 %

04 **Takeaways**
three lessons for developers

05 **General observations**
two protected libraries · what is missing · getting it used

Level 1: Timing & SPA — The Threat

a textbook square-and-multiply, 256-bit modulus, 6-bit exponent — scaled down so one exponentiation fits a capture

Input: Base x, exponent $d = (d_k \ldots d_0)_2$, modulus N
$y \leftarrow 1$;
for $i \leftarrow k$ ***downto*** 0 **do**
  $y \leftarrow (y \cdot y) \bmod N$; `// Square (S)`
  if $d_i == 1$ **then**
    $y \leftarrow (y \cdot x) \bmod N$; `// Multiply (M)`
  end
end
return y;

Algorithm 1: Vulnerable left-to-right $y = x^d \bmod N$

Level 1: The Leak Is the Operation Sequence

$$\mathbf{S} \rightarrow 0 \qquad\qquad \mathbf{S}\,\mathbf{M} \rightarrow 1$$

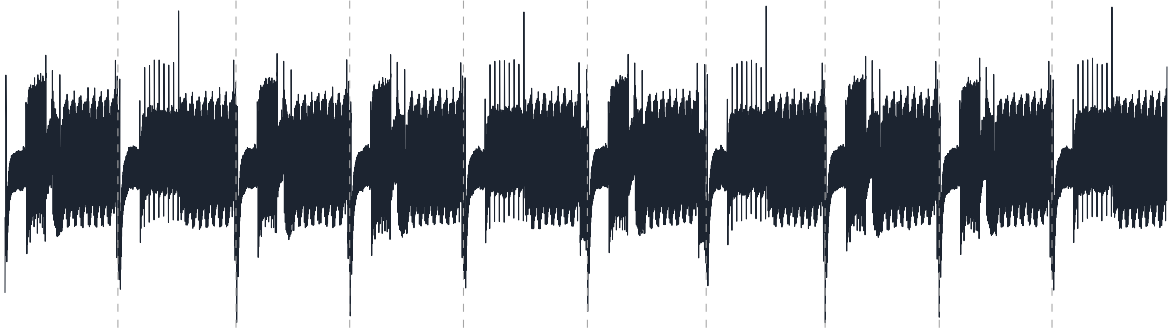
Level 1: The Leak Is the Operation Sequence

S → 0 **S M** → 1

STM32F303, RSA-256: square 2287 cycles, multiply 2434 — **+6.4%**

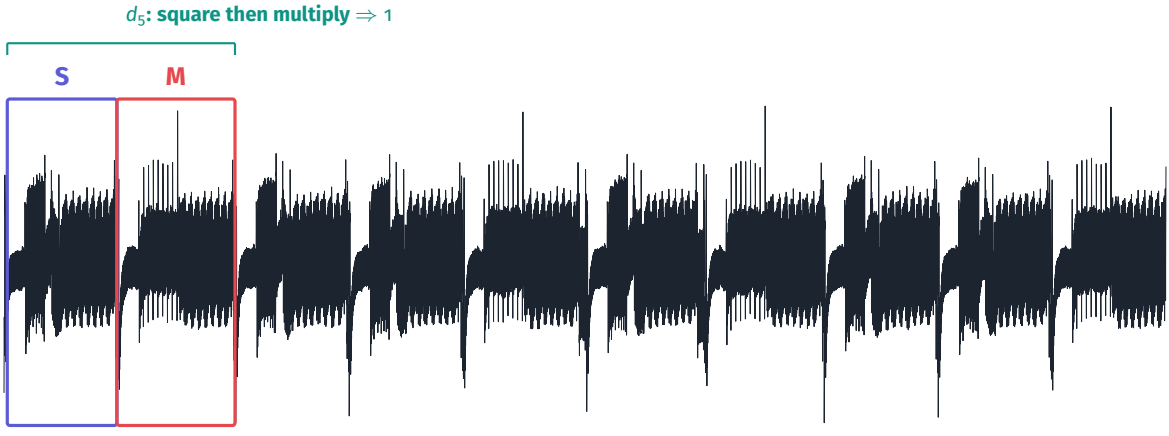
Level 1: SPA — the exponent is in the trace shape

one acquisition, 23417 clock cycles — 10 modular operations, back to back



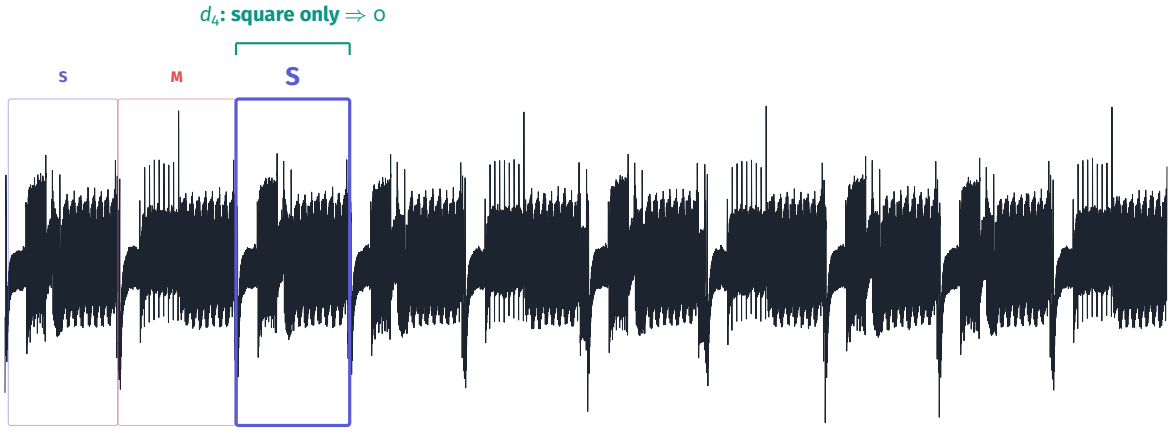
d = ? ? ? ? ? ?

Level 1: SPA — the exponent is in the trace shape



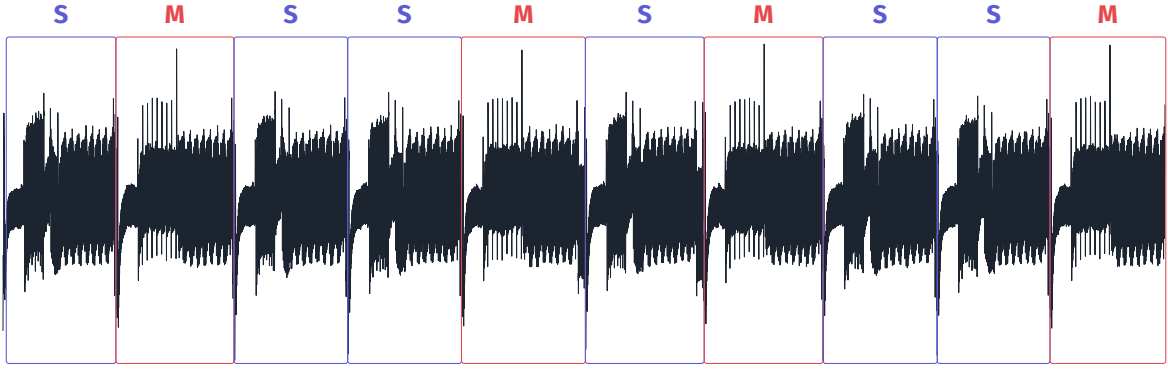
d = 1 ? ? ? ? ?

Level 1: SPA — the exponent is in the trace shape



d = 1 0 ? ? ? ?

Level 1: SPA — the exponent is in the trace shape



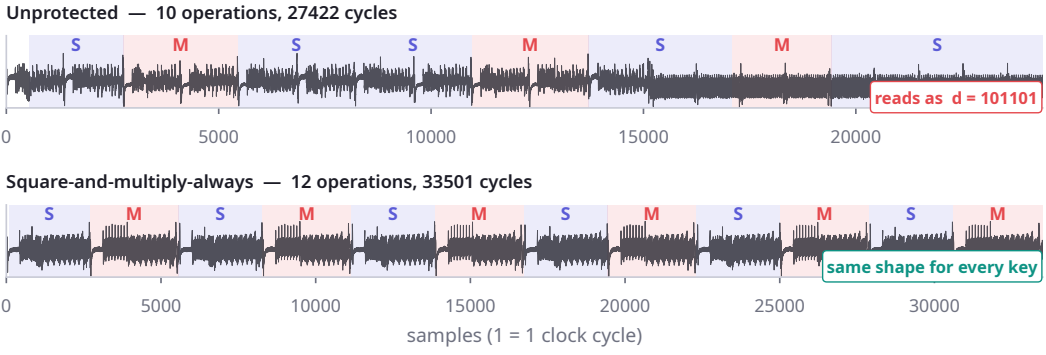
d = 1 0 1 1 0 1

6 exponent bits from a single trace — no key guessing.

Level 1 Fix: Square-and-Multiply-Always

known — textbook

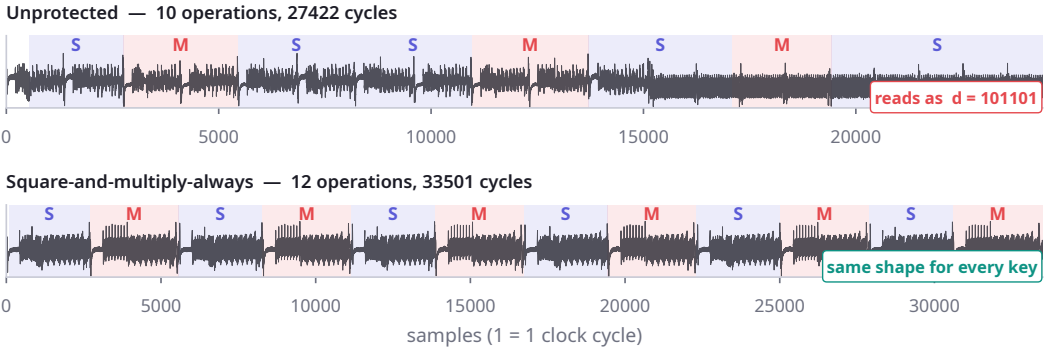
same textbook code, RSA-256, same exponent



Level 1 Fix: Square-and-Multiply-Always

known — textbook

same textbook code, RSA-256, same exponent



Runtime spread across keys: 103.5% → 0

```
mask = 0 - bit;  acc = (acc & ~mask) | (tmp & mask);
```

Timing Pitfall: The Compiler Rewrites Constant-Time Code

constant-time binary GCD, `br_i31_moddiv` — `arm-none-eabi-gcc 14.2, -O2`

Source: masks, no branches

```
/* cAB, cBA, cA: 0 or 1 */
A a_lo -= b_lo & -cAB;
B b_lo -= a_lo & -cBA;
C a_hi ^= (...) & -cA;
D b_lo += b_lo & -cA;
```

Emitted code

```
A mls   r4, ip, r2, r4
B mls   r6, r4, r1, ip
C lsls  r1, r3, #31
C itt   pl
C movpl r7, r2
C movpl r5, ip
D mla   ip, r3, r2, r2
```

GCC inferred 0/1: multiplies, predicated moves — the compiler's choice

Timing Pitfall: The Compiler Rewrites Constant-Time Code

constant-time binary GCD, `br_i31_moddiv` — `arm-none-eabi-gcc 14.2, -O2`

With an optimisation barrier

```
/* cAB, cBA, cA: 0 or 1 */
__asm volatile ("" : "+r"(cA),
      "+r"(cAB), "+r"(cBA));
A a_lo -= b_lo & -cAB;
E b_hi -= a_hi & -cBA;
```

Emitted code

```
A negs  r1, r2
A mov   r2, r1
E negs  r7, r3
E mov   r3, r7
A ldr   r7, [sp]
A and   r2, r2, r7
A sub   r2, lr, r2
E and   r7, r3, r4
E subs  r8, r8, r7
```

Pure ALU masking: negate, AND, subtract — exactly what the source says

Windowed Exponentiation: Fewer Multiplies

Input: Base x, exponent d, window k
// precomputed once, $2^k - 1$ entries
for $u \leftarrow 1$ **to** $2^k - 1$ **do**
  $T[u] \leftarrow x^u \bmod N$;
end
$y \leftarrow 1$;
while *exponent bits remain* **do**
  $b \leftarrow$ next k bits of d;
  for $i \leftarrow 1$ **to** k **do**
    $y \leftarrow y^2 \bmod N$;
  end
  $t \leftarrow T[b]$;
  $y \leftarrow y \cdot t \bmod N$;
end
return y;

Algorithm 2: Fixed-window $y = x^d \bmod N$

Windowed Exponentiation: Fewer Multiplies

Input: Base x, exponent d, window k
// precomputed once, $2^k - 1$ entries
for $u \leftarrow 1$ **to** $2^k - 1$ **do**
  $T[u] \leftarrow x^u \bmod N$;
end
$y \leftarrow 1$;
while *exponent bits remain* **do**
  $b \leftarrow$ next k bits of d;
  for $i \leftarrow 1$ **to** k **do**
    $y \leftarrow y^2 \bmod N$;
  end
  $t \leftarrow T[b]$;
  $y \leftarrow y \cdot t \bmod N$;
end
return y;

Algorithm 3: Fixed-window $y = x^d \bmod N$

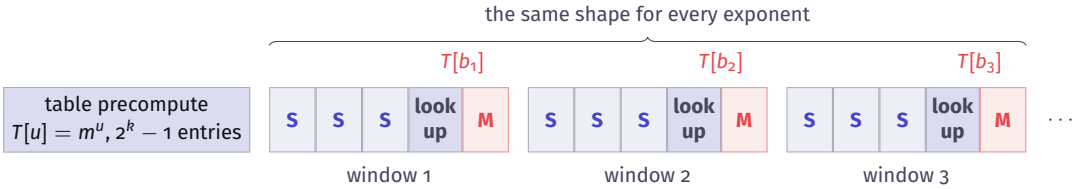
Why a window

k squarings + **1** multiply per k bits
(instead of $k/2$ multiplies)

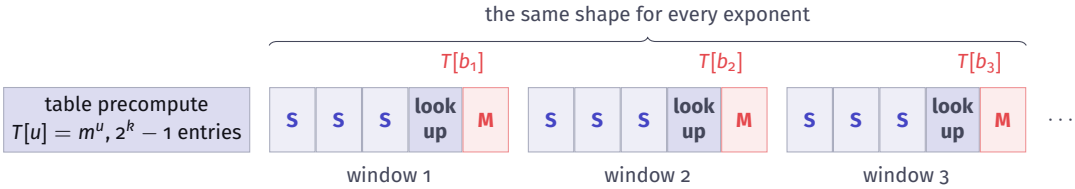
table: $2^k - 1$ entries

BearSSL: $k = 4$ for RSA-4096 CRT

What the Attacker Is Left With



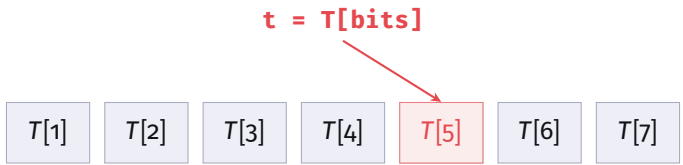
What the Attacker Is Left With



$T[b] = m^b$ — a **secret-indexed value**

The Lookup: Why Not Just `T[b]`

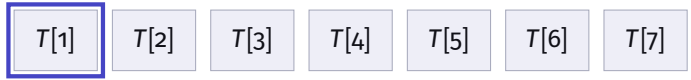
window value $\mathrm{bits} = 5$, $k = 3$



The Lookup: Why Not Just T[b]

window value $\mathrm{bits} = 5$, $k = 3$

mask = 0 — t unchanged

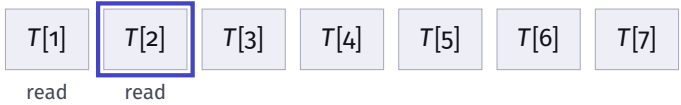


t = —

The Lookup: Why Not Just T[b]

window value $\mathrm{bits} = 5$, $k = 3$

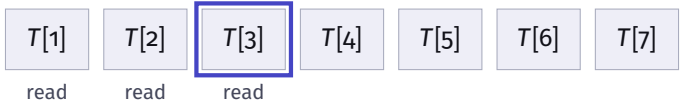
mask = 0 — t unchanged



The Lookup: Why Not Just T[b]

window value $\mathrm{bits} = 5$, $k = 3$

mask = 0 — t unchanged



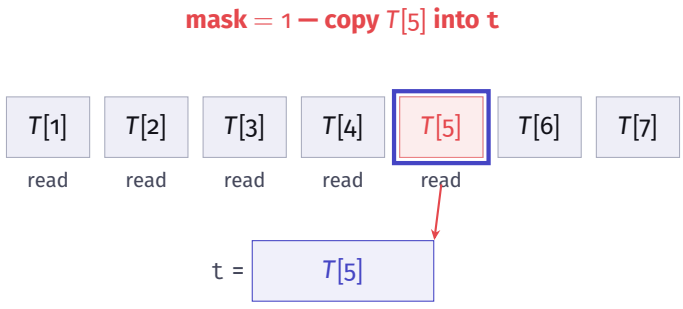
The Lookup: Why Not Just T[b]

window value $\mathrm{bits} = 5$, $k = 3$

mask = 0 — t unchanged

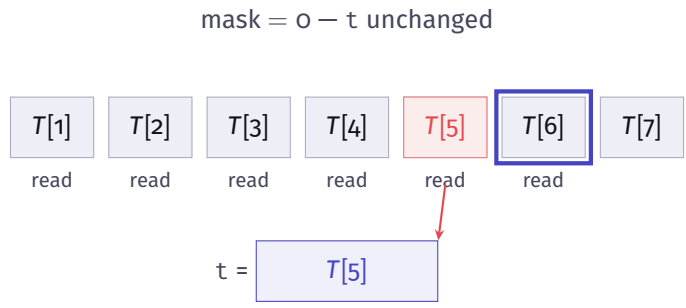
The Lookup: Why Not Just T[b]

window value $\mathrm{bits} = 5$, $k = 3$



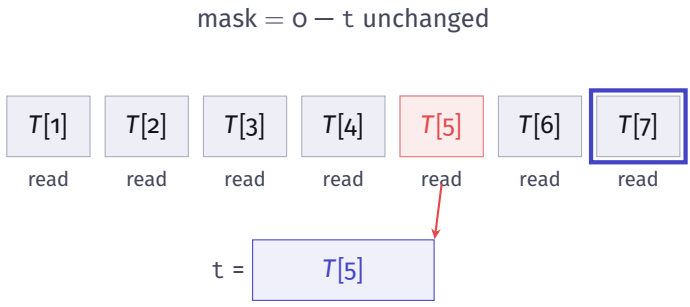
The Lookup: Why Not Just T[b]

window value $\mathrm{bits} = 5$, $k = 3$



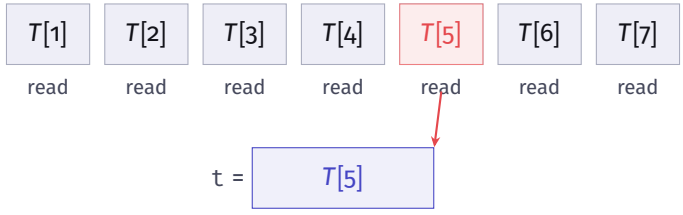
The Lookup: Why Not Just T[b]

window value bits = 5, $k = 3$



The Lookup: Why Not Just `T[b]`

window value $\mathrm{bits} = 5$, $k = 3$



every entry read, same order, same time — **only the mask decides**

```
for (u = 1; u < (1 « k); u++) CCOPY(EQ(u, bits), t, T[u], mlen);
```

Level 2 Fix: Blind the Message and the Exponent

known — in wolfSSL, Mbed TLS

64-bit blinding factors, fresh for every signature

Message blinding

$$M' = M \cdot r_3^{\,e} \bmod N \qquad S = S' \cdot r_3^{\,-1} \bmod N$$

the value in every multiply is random

Kocher, Jaffe, Jun — *Differential Power Analysis*, CRYPTO 1999

Level 2 Fix: Blind the Message and the Exponent

known — in wolfSSL, Mbed TLS

64-bit blinding factors, fresh for every signature

Message blinding

$$M' = M \cdot r_3^{e} \bmod N \qquad S = S' \cdot r_3^{-1} \bmod N$$

the value in every multiply is random

Exponent blinding

$$d'_p = d_p + r_4 \cdot \varphi(p) \qquad d'_q = d_q + r_5 \cdot \varphi(q)$$

the window values are random

Kocher, Jaffe, Jun — *Differential Power Analysis*, CRYPTO 1999

Level 2 Fix: Blind the Message and the Exponent

known — in wolfSSL, Mbed TLS

64-bit blinding factors, fresh for every signature

Message blinding

$$M' = M \cdot r_3^{e} \bmod N \qquad S = S' \cdot r_3^{-1} \bmod N$$

the value in every multiply is random

Exponent blinding

$$d'_p = d_p + r_4 \cdot \varphi(p) \qquad d'_q = d_q + r_5 \cdot \varphi(q)$$

the window values are random

nothing left to predict — **+21 %**

Kocher, Jaffe, Jun — *Differential Power Analysis*, CRYPTO 1999

Level 3: TVLA — A Failed Attack Is Not Yet Evidence

the industry-standard leakage test

The model-free question

Does the **mean** of a sample change with the secret input?

Two groups of traces, one Welch *t*-test per sample:

```
t = ttest_ind(G0, G1, axis=0,
              equal_var=False).statistic
```

No key guess, no power model — one number per sample.

Goodwill, Jun, Jaffe, Rohatgi — NIST NIAT Workshop 2011 · ISO/IEC 17825

Level 3: TVLA — A Failed Attack Is Not Yet Evidence

the industry-standard leakage test

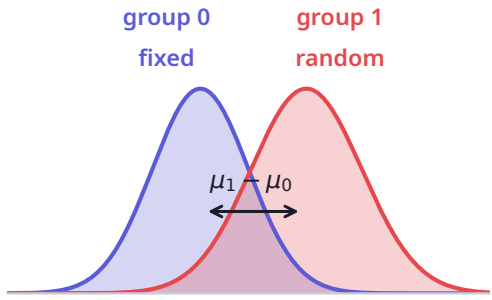
The model-free question

Does the **mean** of a sample change with the secret input?

Two groups of traces, one Welch *t*-test per sample:

```
t = ttest_ind(G0, G1, axis=0,
              equal_var=False).statistic
```

No key guess, no power model — one number per sample.



Goodwill, Jun, Jaffe, Rohatgi — NIST NIAT Workshop 2011 · ISO/IEC 17825

Level 3: Three Groups, One Threshold — $|t| > 6.4$

Husky + CW308 / STM32F3 · PicoScope 3406D · RSA-1024, the largest that fits the capture buffer

1000 traces per group

group 0	fixed message, fixed key
group 1	random message
group 2	random exponent

0 vs 1: message blinding

0 vs 2: exponent blinding

Ding, Zhang, Durvaux, Standaert, Fei — *Towards Sound and Optimal Leakage Detection Procedure*, CARDIS 2017

Level 3: Three Groups, One Threshold — $|t| > 4.5$

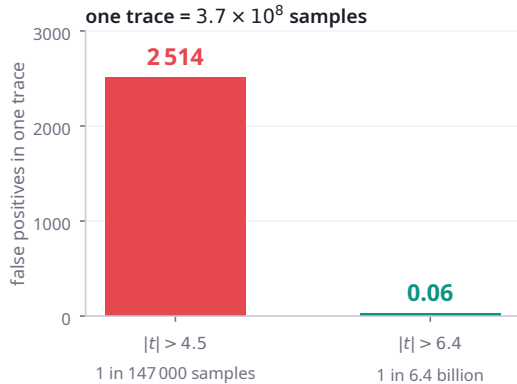
Husky + CW308 / STM32F3 · PicoScope 3406D · RSA-1024, the largest that fits the capture buffer

1000 traces per group

group 0	fixed message, fixed key
group 1	random message
group 2	random exponent

0 vs 1: message blinding

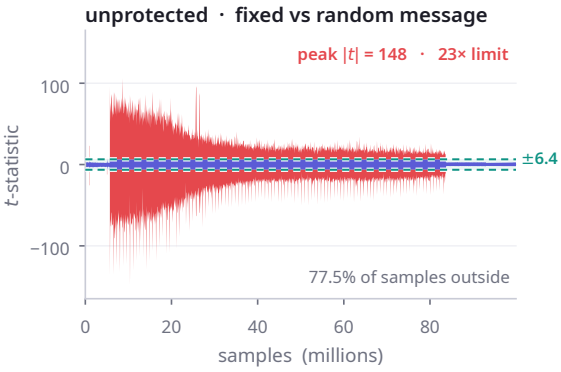
0 vs 2: exponent blinding



Ding, Zhang, Durvaux, Standaert, Fei — *Towards Sound and Optimal Leakage Detection Procedure*, CARDIS 2017

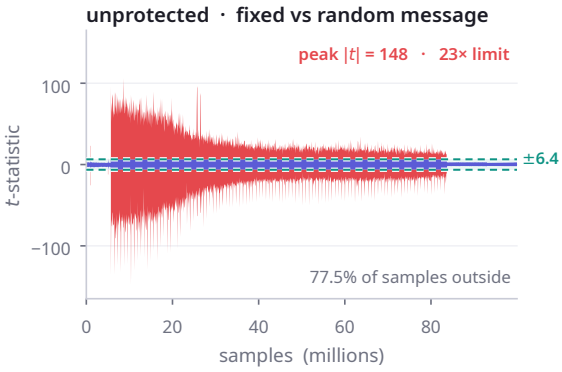
Level 3 Baseline: The Unprotected Path Fails Both Tests

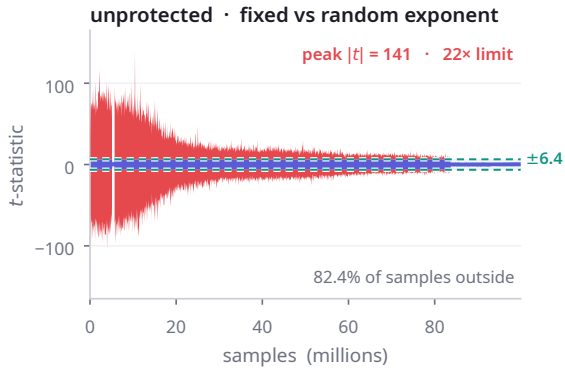
captured: the exponentiation only



Level 3 Baseline: The Unprotected Path Fails Both Tests

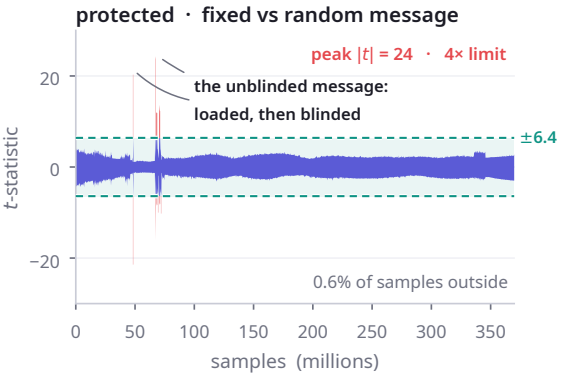
captured: the exponentiation only





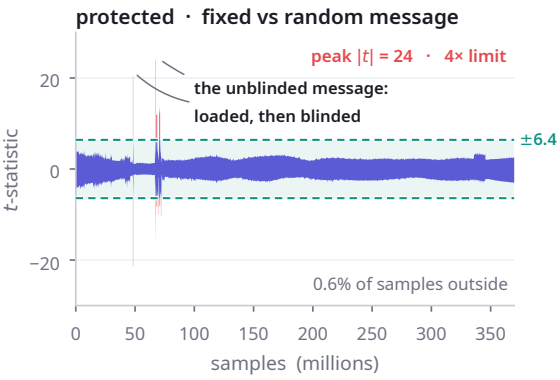
Level 3 Result: Protected — Only the Unblinded Input Is Left

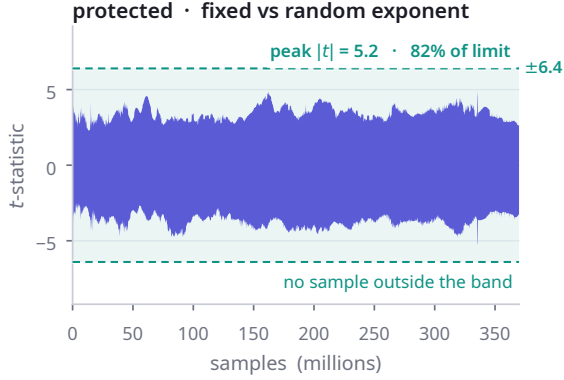
captured: the key update as well as the exponentiation — hence the longer trace



Level 3 Result: Protected — Only the Unblinded Input Is Left

captured: the key update as well as the exponentiation — hence the longer trace





What TVLA Cannot Tell You

One sample, one mean

the *t*-test compares the *mean* of each sample between the two groups — a leak in the variance, or spread over two samples (masking), needs a second-order or multivariate test

What TVLA Cannot Tell You

One sample, one mean

the *t*-test compares the *mean* of each sample between the two groups — a leak in the variance, or spread over two samples (masking), needs a second-order or multivariate test

Only as good as the alignment

late leakage in a 10^8-sample trace is under-reported

What TVLA Cannot Tell You

One sample, one mean

the *t*-test compares the *mean* of each sample between the two groups — a leak in the variance, or spread over two samples (masking), needs a second-order or multivariate test

Only as good as the alignment

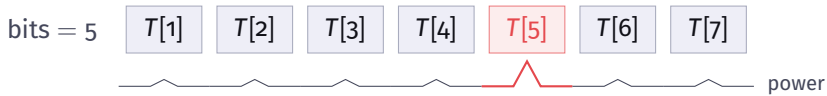
late leakage in a 10^8-sample trace is under-reported

A bound, not a proof

“no first-order leak above 6.4 at 1000 traces per group” — hence the replay

Level 4: Constant-Time Is Not Enough — Address-Bit DPA

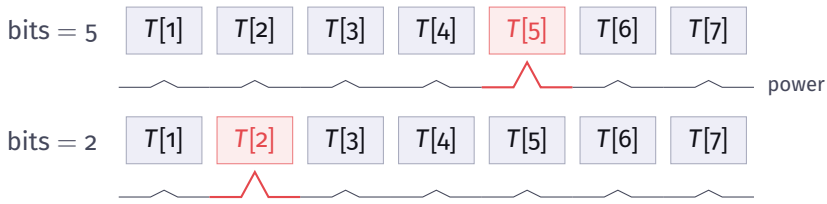
the same sweep, three windows — $k = 3$ shown; BearSSL uses $k = 4$



Itoh, Izu, Takenaka — *Address-Bit Differential Power Analysis*, CHES 2002

Level 4: Constant-Time Is Not Enough — Address-Bit DPA

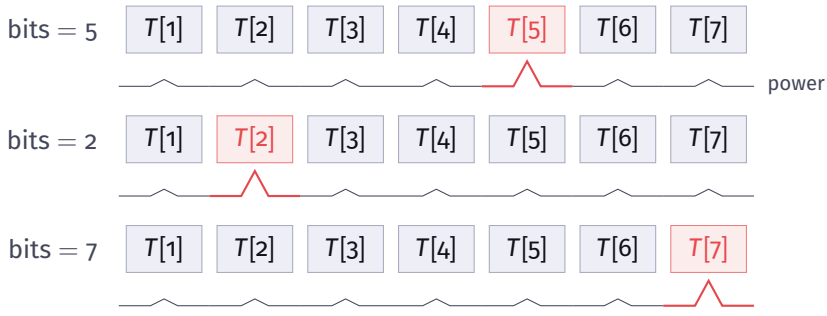
the same sweep, three windows — $k = 3$ shown; BearSSL uses $k = 4$



Itoh, Izu, Takenaka — *Address-Bit Differential Power Analysis*, CHES 2002

Level 4: Constant-Time Is Not Enough — Address-Bit DPA

the same sweep, three windows — $k = 3$ shown; BearSSL uses $k = 4$



every entry read, every time — but **which load reaches t** is the secret

Itoh, Izu, Takenaka — *Address-Bit Differential Power Analysis*, CHES 2002

Level 4 Fix: Permute the Table

known — idea: Itoh et al.

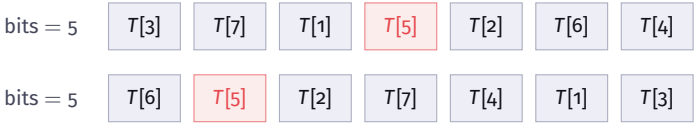
$k = 3$ shown; BearSSL uses $k = 4$

Itoh, Izu, Takenaka — *A Practical Countermeasure against Address-Bit DPA*, CHES 2003

Level 4 Fix: Permute the Table

known — idea: Itoh et al.

$k = 3$ shown; BearSSL uses $k = 4$

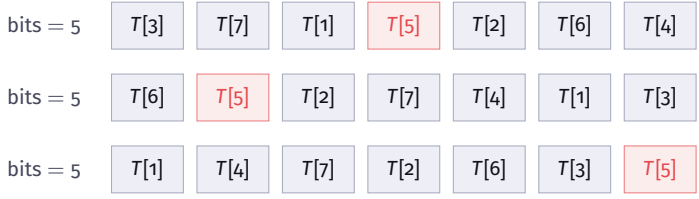


Itoh, Izu, Takenaka — *A Practical Countermeasure against Address-Bit DPA*, CHES 2003

Level 4 Fix: Permute the Table

known — idea: Itoh et al.

$k = 3$ shown; BearSSL uses $k = 4$



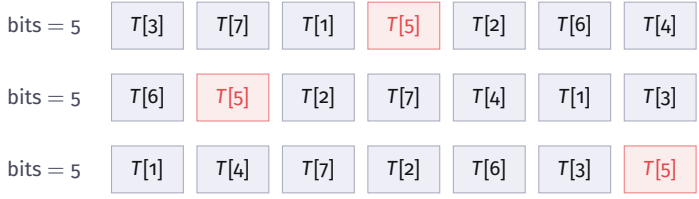
same chunk, a fresh permutation π — the hit lands anywhere

Itoh, Izu, Takenaka — *A Practical Countermeasure against Address-Bit DPA*, CHES 2003

Level 4 Fix: Permute the Table

known — idea: Itoh et al.

$k = 3$ shown; BearSSL uses $k = 4$



same chunk, a fresh permutation π — the hit lands anywhere

entries blinded: $T[u] \leftarrow T[u] + m''$

Itoh, Izu, Takenaka — *A Practical Countermeasure against Address-Bit DPA*, CHES 2003

Level 4: Horizontal — Table Reuse Inside One Trace

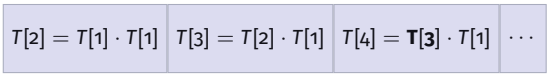


table precompute — order known

one execution, one trace

Level 4: Horizontal — Table Reuse Inside One Trace

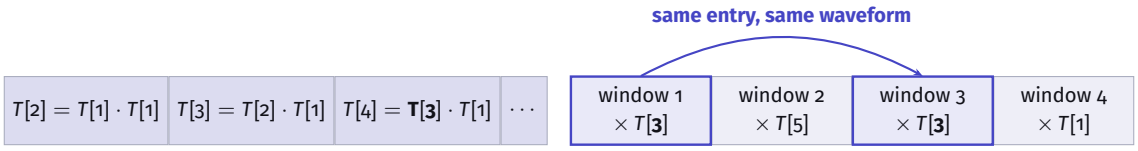
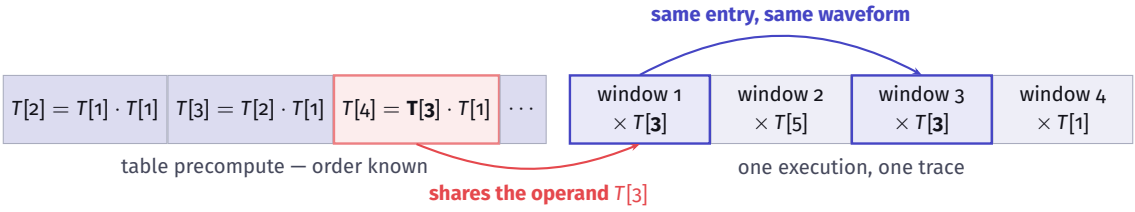


table precompute — order known

one execution, one trace

Level 4: Horizontal — Table Reuse Inside One Trace



Level 4 Fix: Nothing Reused, Nothing Fixed new — this work

Modulus re-randomisation

a fresh random multiple of the modulus, $m'' = m' \cdot r$, every window — inside one fixed Montgomery domain

Level 4 Fix: Nothing Reused, Nothing Fixed new — this work

Modulus re-randomisation

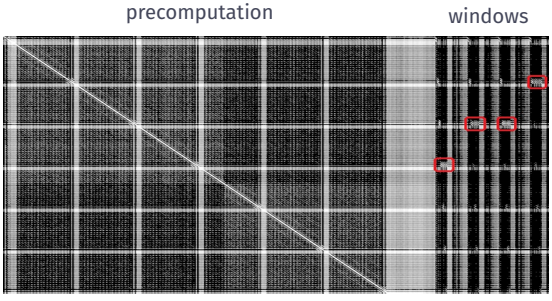
a fresh random multiple of the modulus, $m'' = m' \cdot r$, every window — inside one fixed Montgomery domain

Pre-randomised 11-tuple key

$\varphi(p)'$, $\varphi(q)'$ blinded, refreshed every signature

Level 4: Cross-Correlation — Unprotected

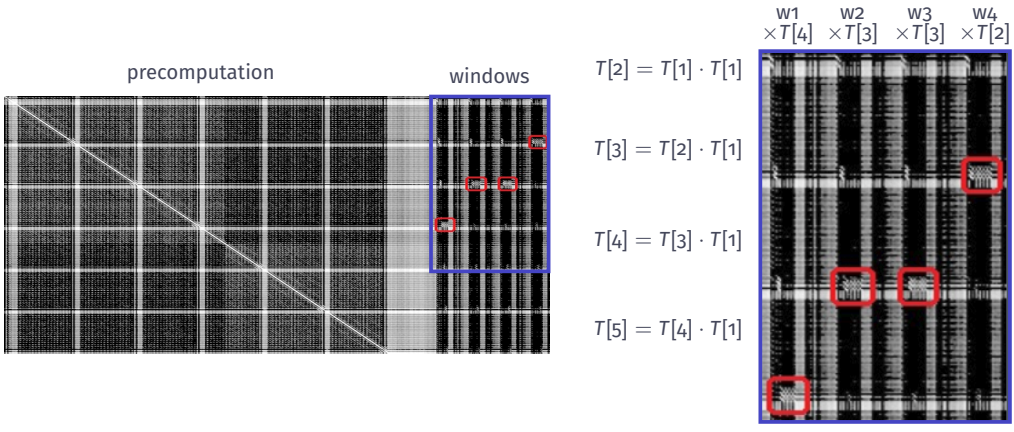
1000 traces, exponent blinding off; chunks $4, 3, 3, 2$; $k = 3$



every pair of sub-traces in one execution, correlated across the traces

Level 4: Cross-Correlation — Unprotected

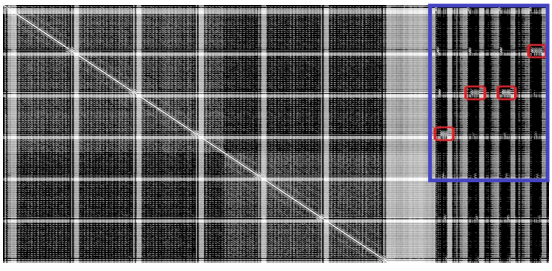
1000 traces, exponent blinding off; chunks $4, 3, 3, 2$; $k = 3$



zoom: the window multiplies against the precomputation steps

Level 4: Cross-Correlation — Unprotected

1000 traces, exponent blinding off; chunks $4, 3, 3, 2$; $k = 3$

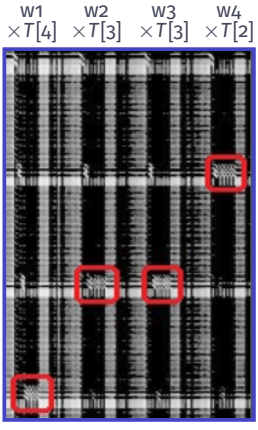


$T[2] = T[1] \cdot T[1]$

$T[3] = T[2] \cdot T[1]$

$T[4] = T[3] \cdot T[1]$

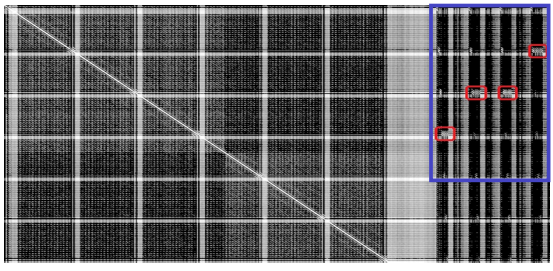
$T[5] = T[4] \cdot T[1]$



w1 multiplies by $T[4]$ — so did the step that built $T[5]$

Level 4: Cross-Correlation — Unprotected

1000 traces, exponent blinding off; chunks $4, 3, 3, 2$; $k = 3$

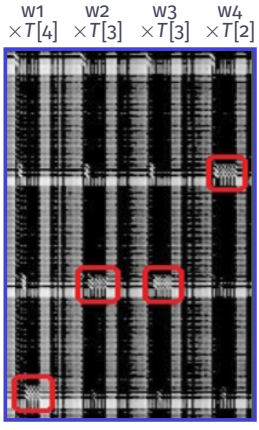


$T[2] = T[1] \cdot T[1]$

$T[3] = T[2] \cdot T[1]$

$T[4] = T[3] \cdot T[1]$

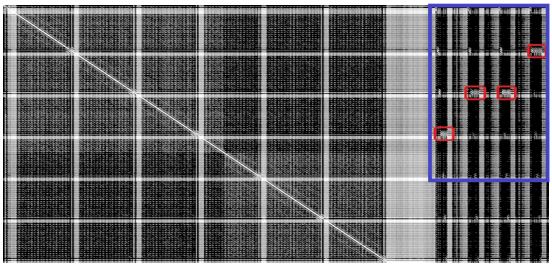
$T[5] = T[4] \cdot T[1]$



w2, w3 multiply by $T[3]$ — so did the step that built $T[4]$

Level 4: Cross-Correlation — Unprotected

1000 traces, exponent blinding off; chunks $4, 3, 3, 2$; $k = 3$

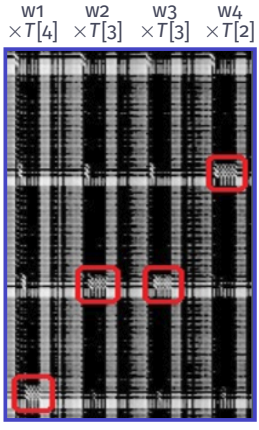


$T[2] = T[1] \cdot T[1]$

$T[3] = T[2] \cdot T[1]$

$T[4] = T[3] \cdot T[1]$

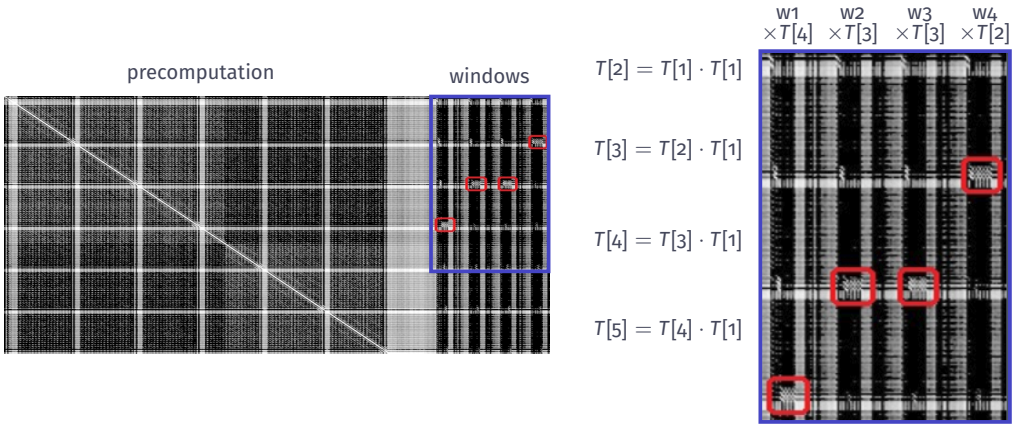
$T[5] = T[4] \cdot T[1]$



w4 multiplies by $T[2]$ — so did the step that built $T[3]$

Level 4: Cross-Correlation — Unprotected

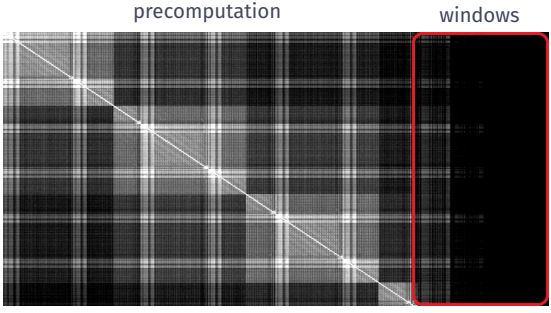
1000 traces, exponent blinding off; chunks $4, 3, 3, 2$; $k = 3$



a shared operand lights up — the exponent chunks, read off the trace

Level 4: Cross-Correlation — Protected

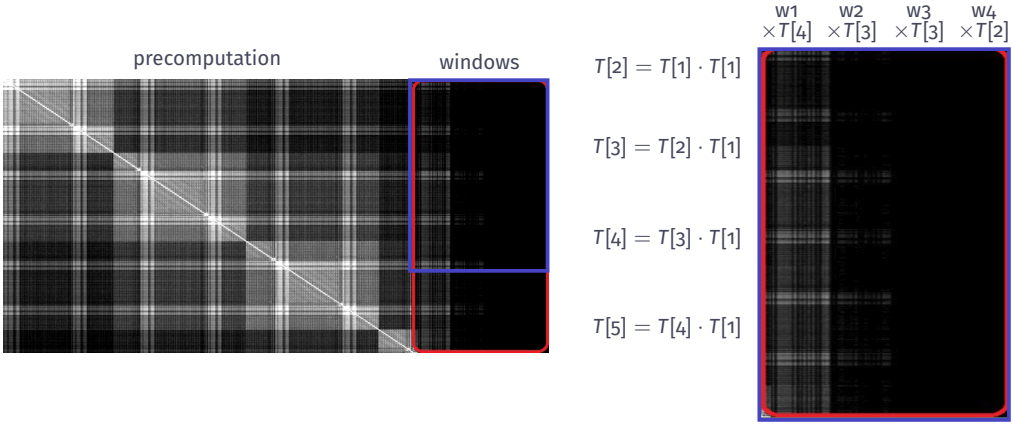
same analysis, same exponent chunks — modulus re-randomised, table permuted



the same analysis on the hardened implementation

Level 4: Cross-Correlation — Protected

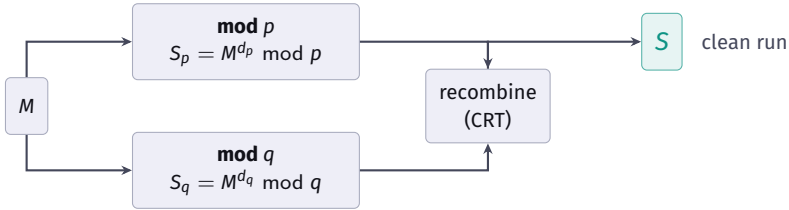
same analysis, same exponent chunks — modulus re-randomised, table permuted



the same region: nothing lights up — no shared operand left to see

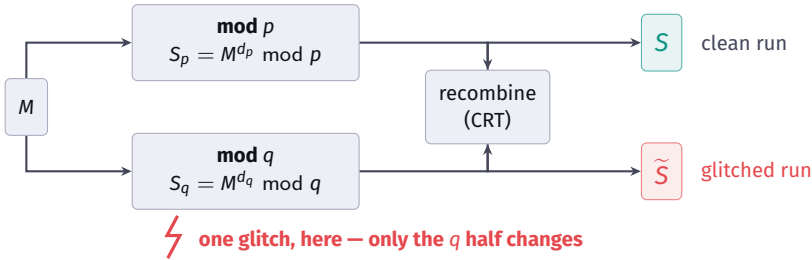
Level 5: Fault Injection — One Fault Is Enough (Bellcore)

RSA-CRT: two half-size exponentiations, recombined — about four times faster



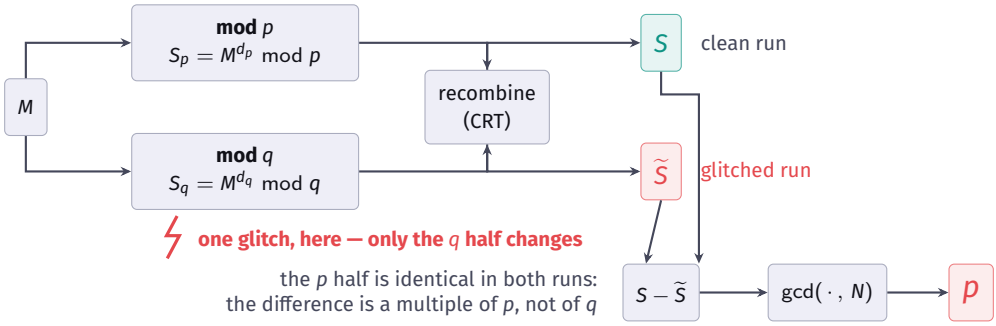
Level 5: Fault Injection — One Fault Is Enough (Bellcore)

RSA-CRT: two half-size exponentiations, recombined — about four times faster



Level 5: Fault Injection — One Fault Is Enough (Bellcore)

RSA-CRT: two half-size exponentiations, recombined — about four times faster



Level 5 Fix: Two Stages, No Branch

known — each stage **new — both in one library**

1 — Infective computation

checksums over the recombination; a fault corrupts the unblinding factor → noise

Ciet, Joye — *Practical Fault Countermeasures for Chinese Remaindering Based RSA*, FDTC 2005

Level 5 Fix: Two Stages, No Branch

known — each stage | **new — both in one library**

1 — Infective computation

checksums over the recombination; a fault corrupts the unblinding factor → noise

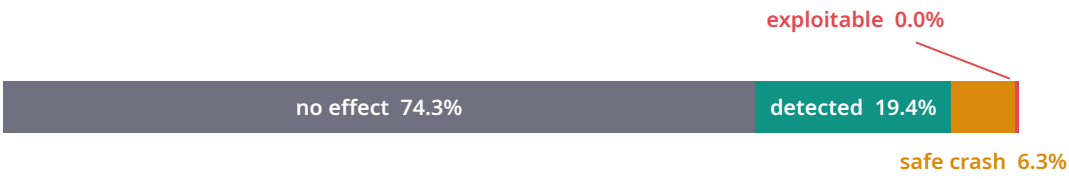
2 — Verification

$S^e \bmod N \stackrel{?}{=} M$ — a branchless mask zeroes the output

Ciet, Joye — *Practical Fault Countermeasures for Chinese Remaindering Based RSA*, FDTC 2005

Level 5: Fault Injection Verification Campaign

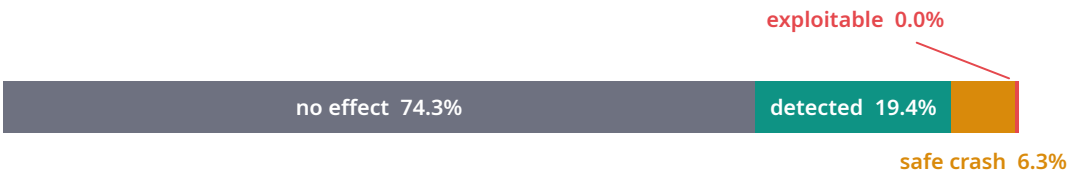
`fault-finder`, a software simulator — every 1–3-instruction skip in BearSSL RSA-CRT, 143 565 423 runs



Murdock, Thompson, Oswald — *FaultFinder*, ASHES 2024

Level 5: Fault Injection Verification Campaign

`fault-finder`, a software simulator — every 1–3-instruction skip in BearSSL RSA-CRT, 143 565 423 runs



exploitable: 0.0 % — zero keys factored

Murdock, Thompson, Oswald — *FaultFinder*, ASHES 2024

Outline

01 **Method**
the hardening cycle · the testing pipeline · ChipWhisperer

02 **Five levels**
timing & SPA · blinding · TVLA, the gate · table position & reuse · faults

03 **Integration**
what is new · the layered architecture in BearSSL · what it costs: +56 %

04 **Takeaways**
three lessons for developers

05 **General observations**
two protected libraries · what is missing · getting it used

What Is New — and What It Builds On

the approach follows sca25519 (Curve25519)

Known — reused

constant-time code
message & exponent blinding
table randomisation
infective computation, verification
TVLA

Batina et al. — *SoK: SCA-Secure ECC in Software*, TCHES 2023

What Is New — and What It Builds On

the approach follows sca25519 (Curve25519)

Known — reused	New
constant-time code	**modulus re-randomisation**
message & exponent blinding	**pre-randomised key**
table randomisation	**both FI stages in one library**
infective computation, verification	
TVLA	

Batina et al. — *SoK: SCA-Secure ECC in Software*, TCHES 2023

What Is New — and What It Builds On

the approach follows sca25519 (Curve25519)

Known — reused

constant-time code
message & exponent blinding
table randomisation
infective computation, verification
TVLA

New

modulus re-randomisation
pre-randomised key
both FI stages in one library

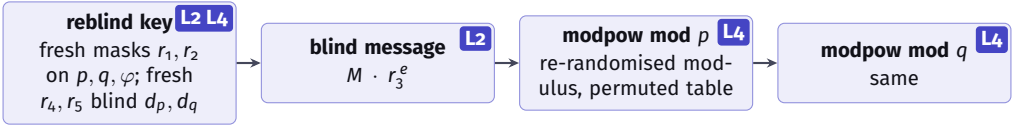
The engineering

full stack, open source, in BearSSL
measured end to end
+56 %

Batina et al. — *SoK: SCA-Secure ECC in Software*, TCHES 2023

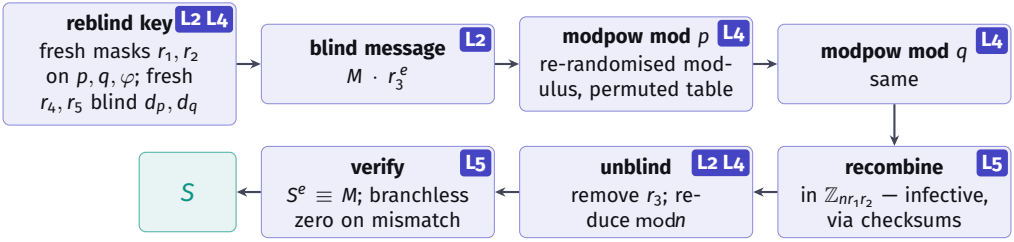
The Layered Architecture in BearSSL: One Protected Signature

stock BearSSL: modpow p, modpow q, recombine — everything shaded is added



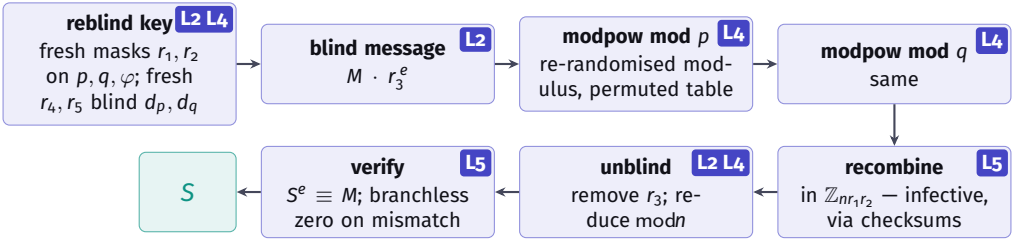
The Layered Architecture in BearSSL: One Protected Signature

stock BearSSL: modpow p, modpow q, recombine — everything shaded is added



The Layered Architecture in BearSSL: One Protected Signature

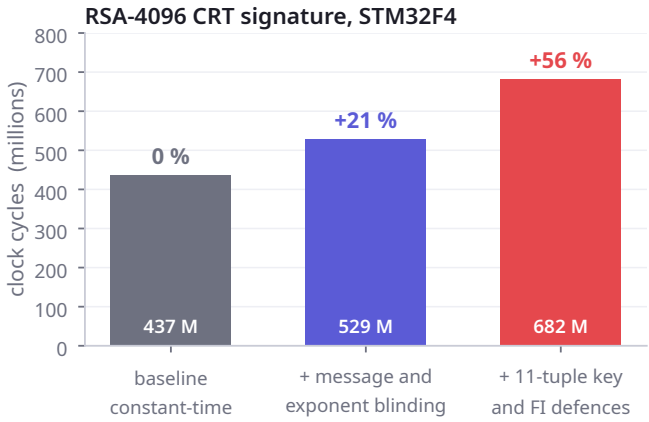
stock BearSSL: modpow p, modpow q, recombine — everything shaded is added



Key	Arithmetic	Platform
5-tuple → 11-tuple, +700 B	values 64 bits longer — re-checked	RNG · 6.6 KB · GCC 14.2 -O2

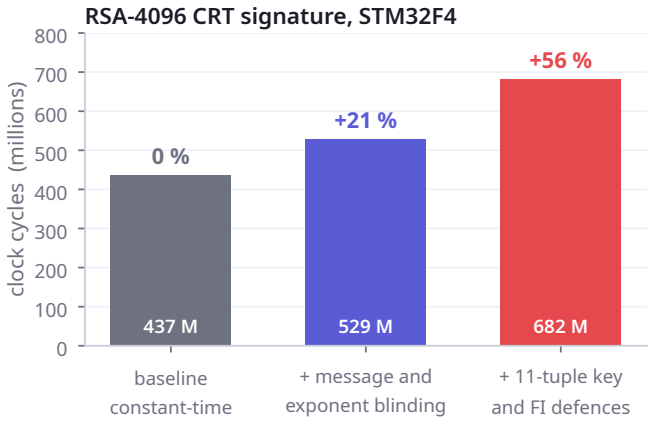
Performance Trade-Offs

measured on STM32F4; the leakage work used STM32F3 — both Cortex-M4



Performance Trade-Offs

measured on STM32F4; the leakage work used STM32F3 — both Cortex-M4



Memory: key structure + ∼700 B (55 %), workspace 6.6 KB — under 3.5 % of the SRAM.

Outline

01 **Method**
the hardening cycle · the testing pipeline · ChipWhisperer

02 **Five levels**
timing & SPA · blinding · TVLA, the gate · table position & reuse · faults

03 **Integration**
what is new · the layered architecture in BearSSL · what it costs: +56 %

04 **Takeaways**
three lessons for developers

05 **General observations**
two protected libraries · what is missing · getting it used

Takeaways for Developers

1 Constant-time is a property of the binary

pin the toolchain · barrier the masks · check what ships — learned on: Timing Pitfall

Takeaways for Developers

1 Constant-time is a property of the binary

pin the toolchain · barrier the masks · check what ships — learned on: Timing Pitfall

2 Blinded values are longer

re-check every routine for $|N| + 64$ bits — learned on: The Layered Architecture

Takeaways for Developers

1 Constant-time is a property of the binary

pin the toolchain · barrier the masks · check what ships — learned on: Timing Pitfall

2 Blinded values are longer

re-check every routine for $|N| + 64$ bits — learned on: The Layered Architecture

3 “Done” has one meaning

the original attack now fails · nothing above 6.4 unexplained — learned on: The Testing Pipeline

Outline

01 **Method**
the hardening cycle · the testing pipeline · ChipWhisperer

02 **Five levels**
timing & SPA · blinding · TVLA, the gate · table position & reuse · faults

03 **Integration**
what is new · the layered architecture in BearSSL · what it costs: +56 %

04 **Takeaways**
three lessons for developers

05 **General observations**
two protected libraries · what is missing · getting it used

Two Protected Libraries, Built the Hard Way

RSA and ECC — the same layered approach, both open source

Protected BearSSL RSA

this talk — RSA-CRT on Cortex-M4 —
github.com/sca-protected-rsa-library

sca25519

X25519 on μNaCl —
github.com/sca-secure-library-sca25519

Two Protected Libraries, Built the Hard Way

RSA and ECC — the same layered approach, both open source

Protected BearSSL RSA

this talk — RSA-CRT on Cortex-M4 —
github.com/sca-protected-rsa-library

sca25519

X25519 on μNaCl —
github.com/sca-secure-library-sca25519

Hard to publish

every claim needs this much evaluation

Hard to maintain

post-publication maintenance

What the Ecosystem Is Missing

Honest claims	Protected hardware	Open alternatives
some popular libraries overstate their security	little to none	commercial solutions: closed, Common Criteria

What the Ecosystem Is Missing

Honest claims

some popular libraries overstate their security

Protected hardware

little to none

Open alternatives

commercial solutions: closed, Common Criteria

Documentation

implementation choices rarely explained

Whole primitives

no SCA-protected SHA-2 is available

Getting Protected Code Used

The hardest part

making people use the libraries

Getting Protected Code Used

The hardest part

making people use the libraries

What we will publish

all evaluation code — ChipWhisperer scripts (SCA) and the fault-simulation setup (FI)

Getting Protected Code Used

The hardest part

making people use the libraries

What we will publish

all evaluation code — ChipWhisperer scripts (SCA) and the fault-simulation setup (FI)

Thank you!

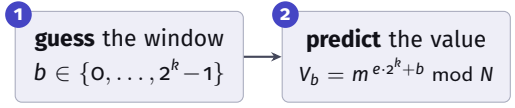
Appendix

CPA on the window table — the attack that motivated Level 2

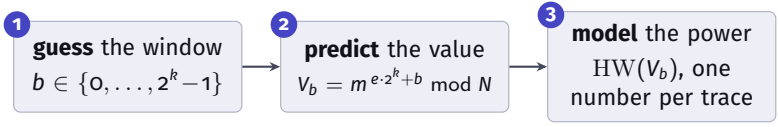
Appendix: CPA — Guessing One Window at a Time

1. **guess** the window $b \in \{0, \ldots, 2^k - 1\}$

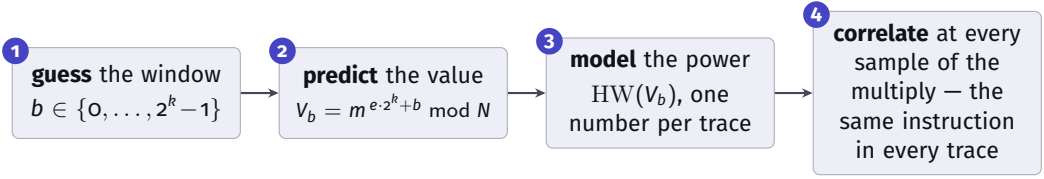
Appendix: CPA — Guessing One Window at a Time



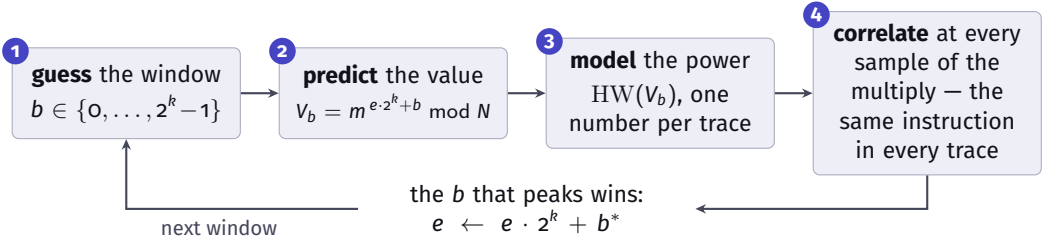
Appendix: CPA — Guessing One Window at a Time



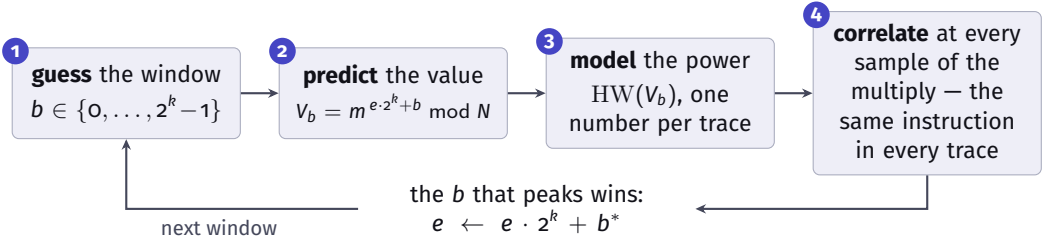
Appendix: CPA — Guessing One Window at a Time



Appendix: CPA — Guessing One Window at a Time



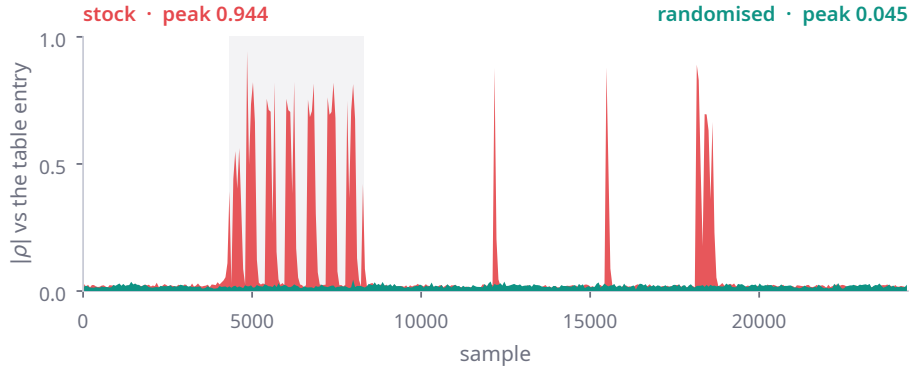
Appendix: CPA — Guessing One Window at a Time



16×512 guesses for RSA-4096 CRT — **additive, not exponential**

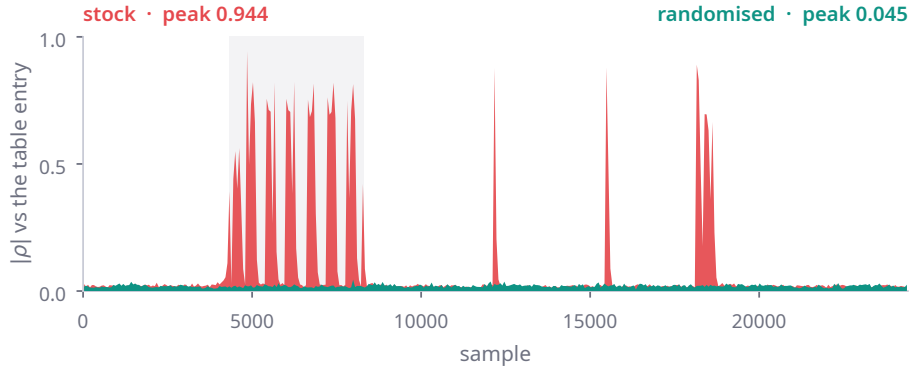
Appendix: CPA — The Table Entry Is Visible, Until It Is Blinded

BearSSL's windowed path at RSA-64, $k = 3$, a three-window exponent, 10 000 traces — scaled down so one exponentiation fits a capture



Appendix: CPA — The Table Entry Is Visible, Until It Is Blinded

BearSSL's windowed path at RSA-64, $k = 3$, a three-window exponent, 10 000 traces — scaled down so one exponentiation fits a capture



stock: 3/3 windows recovered **randomised: 0.045 vs a 0.040 noise floor**

Appendix: CPA — More Traces Do Not Help