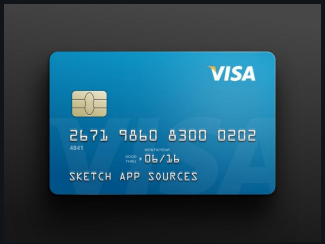
Practical timing and SEMA on embedded OpenSSL’s ECDSA

Julien Eynard, Guénaël Renault, Franck Rondepierre, Adrian Thillard

Security of crypto libs against practical attacks

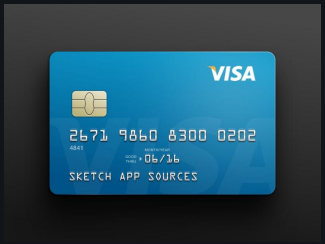
Crypto is implemented:

- on your PC
- on your phone
- on your smart cards
- on your embedded devices



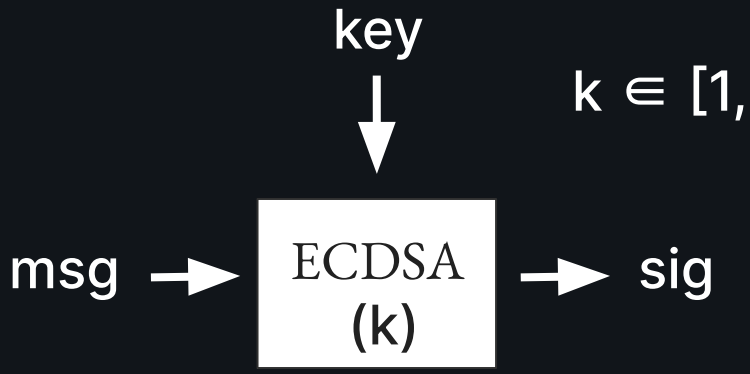
Security of crypto libs against practical attacks

Crypto is implemented:
- on your PC
- on your phone
- on your smart cards
- on your embedded devices

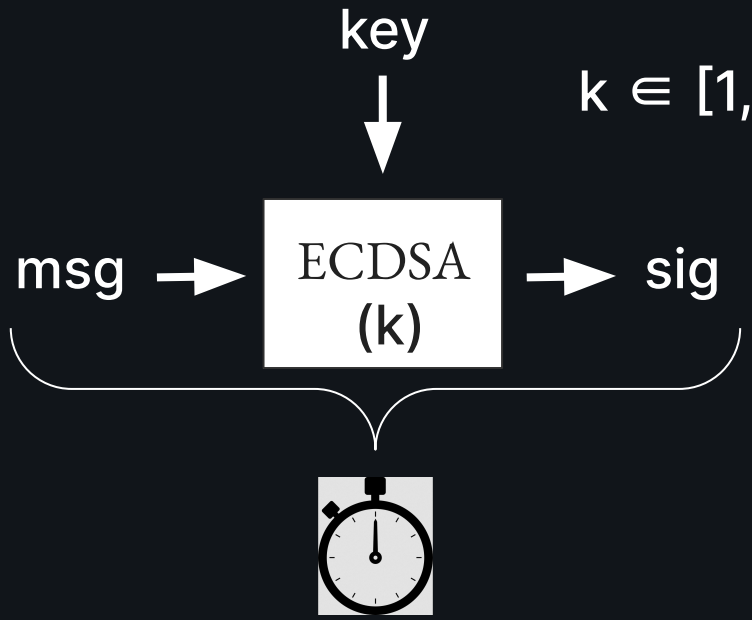


Attackers are practical:
- timing attacks
- cache attacks
- SCA attacks

Timing attacks: Minerva / TPM.fail

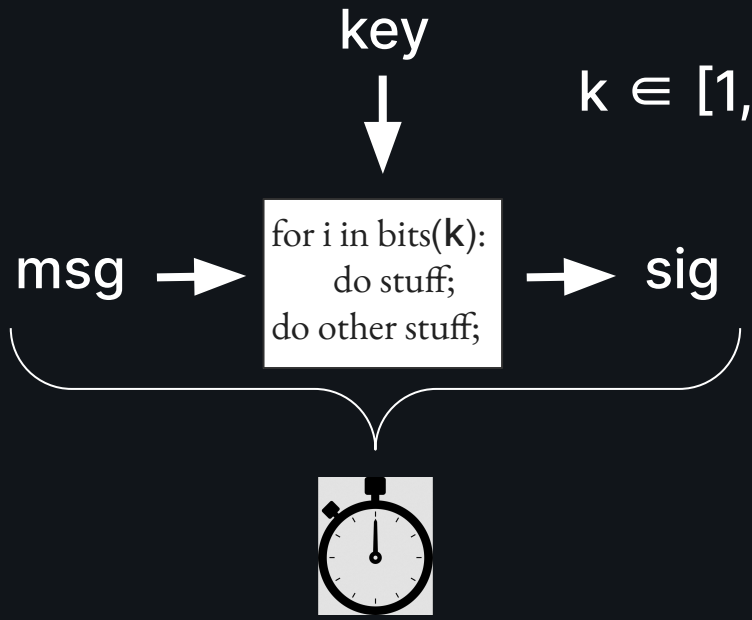


Timing attacks: Minerva / TPM.fail



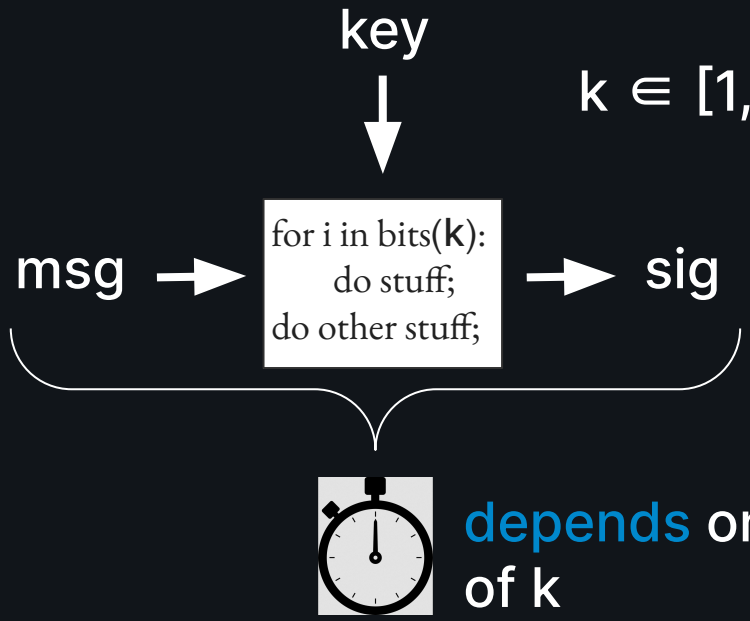
Minerva

Timing attacks: Minerva / TPM.fail



Minerva

Timing attacks: Minerva / TPM.fail



Minerva

depends on the bit length of k

Several sig with info on k allows to recover key (LLL)
(more info in notebook!)

OpenSSL Threat Model

Threat Model

Certain threats are currently considered outside of the scope of the OpenSSL threat model. Accordingly, we do not consider OpenSSL secure against the following classes of attacks:

- same physical system side channel
- CPU/hardware flaws
- physical fault injection
- physical observation side channels (e.g. power consumption, EM emissions, etc)

Mitigations for security issues outside of our threat scope may still be addressed, however we do not class these as OpenSSL vulnerabilities and will therefore not issue CVEs for any mitigations to address these issues.

We are working towards making the same physical system side channel attacks very hard.

Prior to the threat model being included in this policy, CVEs were sometimes issued for these classes of attacks. The existence of a previous CVE does not override this policy going forward.

OpenSSL Threat Model

Threat Model

Certain threats are currently considered outside of the scope of the OpenSSL threat model. Accordingly, we do not consider OpenSSL secure against the following classes of attacks:

- same physical system side channel
- CPU/hardware flaws
- physical fault injection
- physical observation side channels (e.g. power consumption, EM emissions, etc)

Mitigations for security issues outside of our threat scope may still be addressed, however we do not class these as OpenSSL vulnerabilities and will therefore not issue CVEs for any mitigations to address these issues.

We are working towards making the same physical system side channel attacks very hard.

Prior to the threat model being included in this policy, CVEs were sometimes issued for these classes of attacks. The existence of a previous CVE does not override this policy going forward.

So, what security does OpenSSL provide in an embedded setting?

Safe or not safe? A look at ECDSA

```
145 int ossl_ec_scalar_mul_ladder(const EC_GROUP *group, EC_POINT *r,
146                               const BIGNUM *scalar, const EC_POINT *point,
147                               BN_CTX *ctx)
```

```
116 /*-
117  * This functions computes a single point multiplication over the EC group,
118  * using, at a high level, a Montgomery ladder with conditional swaps, with
119  * various timing attack defenses.
120  *
121  * It performs either a fixed point multiplication
122  *          (scalar * generator)
123  * when point is NULL, or a variable point multiplication
124  *          (scalar * point)
125  * when point is not NULL.
126  *
127  * `scalar` cannot be NULL and should be in the range [0,n) otherwise all
128  * constant time bets are off (where n is the cardinality of the EC group).
129  *
130  * This function expects `group->order` and `group->cardinality` to be well
131  * defined and non-zero: it fails with an error code otherwise.
132  *
133  * NB: This says nothing about the constant-timeness of the ladder step
134  * implementation (i.e., the default implementation is based on EC_POINT_add and
135  * EC_POINT_dbl, which of course are not constant time themselves) or the
136  * underlying multiprecision arithmetic.
```

Safe or not safe? A look at ECDSA

```
145  int ossl_ec_scalar_mul_ladder(const EC_GROUP *group, EC_POINT *r,
146                                const BIGNUM *scalar, const EC_POINT *point,
147                                BN_CTX *ctx)
```

```
116  /*-
117   * This functions computes a single point multiplication over the EC group,
118   * using, at a high level, a Montgomery ladder with conditional swaps, with
119   * various timing attack defenses.
120   *
121   * It performs either a fixed point multiplication
122   *          (scalar * generator)
123   * when point is NULL, or a variable point multiplication
124   *          (scalar * point)
125   * when point is not NULL.
126   *
127   * `scalar` cannot be NULL and should be in the range [0,n) otherwise all
128   * constant time bets are off (where n is the cardinality of the EC group).
129   *
130   * This function expects `group->order` and `group->cardinality` to be well
131   * defined and non-zero: it fails with an error code otherwise.
132   *
133   * NB: This says nothing about the constant-timeness of the ladder step
134   * implementation (i.e., the default implementation is based on EC_POINT_add and
135   * EC_POINT_dbl, which of course are not constant time themselves) or the
136   * underlying multiprecision arithmetic.
```

Classic math trick:

$k' \in [1, 2^{64} n - 1]$

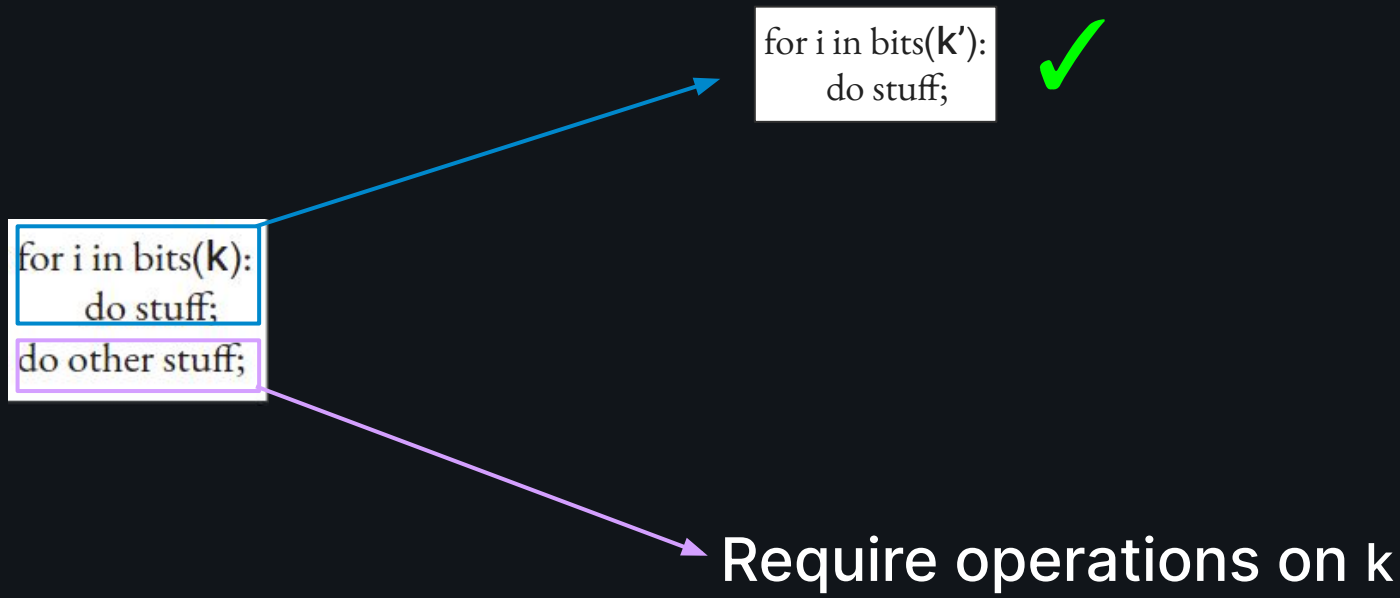
⇒ loops are longer, but the length of $k = k' \bmod n$ is hidden

https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5-draft.pdf

A.3.1 Per-Message Secret Number Generation Using Extra Random Bits

This method uses a cryptographically strong RBG to produce a random bit string that is at least 64 bits longer than the bit-size of the requested random integer k in the interval $[1, n-1]$. More bits are requested from the RBG than are needed for k so that statistical bias introduced by the modular reduction step is negligible.

Another flaw?



Another flaw

```
1  int bn_mul_mont_fixed_top(BIGNUM *r, const BIGNUM *a, const BIGNUM
       *b, BN_MONT_CTX *mont, BN_CTX *ctx)
2  {
3      BIGNUM *tmp;
4      int ret = 0;
5      int num = mont->N.top;
6
7      if (num > 1 && a->top == num && b->top == num) {
8          if (bn_wexpand(r, num) == NULL)
9              return 0;
10         if (bn_mul_mont(r->d, a->d, b->d, mont->N.d, mont->n0, num)
       ) {
11             r->neg = a->neg ^ b->neg;
12             r->top = num;
13             r->flags |= BN_FLG_FIXED_TOP;
14             return 1;
15         }
16     }
17     [...]
18     if (a == b) {
19         if (!bn_sqr_fixed_top(tmp, a, ctx))
20             goto err;
21     } else {
22         if (!bn_mul_fixed_top(tmp, a, b, ctx))
23             goto err;
24     }
25     /* reduce from aRR to aR */
26     if (!bn_from_montgomery_word(r, tmp, mont))
27         goto err;
28     ret = 1;
29 err:
30     BN_CTX_end(ctx);
31     return ret;
32 }
```

$k \in [1,n-1]$

if (nb_words(**n**) == nb_words(**k**)):

Another flaw

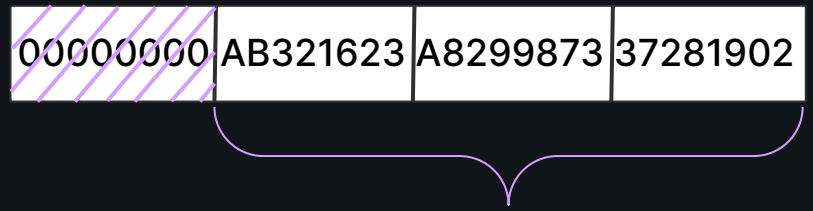
```
int bn_mul_mont_fixed_top(BIGNUM *r, const BIGNUM *a, const BIGNUM
    *b, BN_MONT_CTX *mont, BN_CTX *ctx)
{
    BIGNUM *tmp;
    int ret = 0;
    int num = mont->N.top;

    if (num > 1 && a->top == num && b->top == num) {
        if (bn_wexpand(r, num) == NULL)
            return 0;
        if (bn_mul_mont(r->d, a->d, b->d, mont->N.d, mont->n0, num)
    ) {
            r->neg = a->neg ^ b->neg;
            r->top = num;
            r->flags |= BN_FLG_FIXED_TOP;
            return 1;
        }
    }
    [...]
    if (a == b) {
        if (!bn_sqr_fixed_top(tmp, a, ctx))
            goto err;
    } else {
        if (!bn_mul_fixed_top(tmp, a, b, ctx))
            goto err;
    }
    /* reduce from aRR to aR */
    if (!bn_from_montgomery_word(r, tmp, mont))
        goto err;
    ret = 1;
 err:
    BN_CTX_end(ctx);
    return ret;
}
```

$k \in [1,n-1]$

$\text{if } (\text{nb\_words}(n) == \text{nb\_words}(k)):$

OpenSSL encodes a bignum on the lowest necessary number of words



Another flaw

```
int bn_mul_mont_fixed_top(BIGNUM *r, const BIGNUM *a, const BIGNUM
    *b, BN_MONT_CTX *mont, BN_CTX *ctx)
{
    BIGNUM *tmp;
    int ret = 0;
    int num = mont->N.top;

    if (num > 1 && a->top == num && b->top == num) {
        if (bn_wexpand(r, num) == NULL)
            return 0;
        if (bn_mul_mont(r->d, a->d, b->d, mont->N.d, mont->n0, num)
    ) {
            r->neg = a->neg ^ b->neg;
            r->top = num;
            r->flags |= BN_FLG_FIXED_TOP;
            return 1;
        }
    }
    [...]
    if (a == b) {
        if (!bn_sqr_fixed_top(tmp, a, ctx))
            goto err;
    } else {
        if (!bn_mul_fixed_top(tmp, a, b, ctx))
            goto err;
    }
    /* reduce from aRR to aR */
    if (!bn_from_montgomery_word(r, tmp, mont))
        goto err;
    ret = 1;
err:
    BN_CTX_end(ctx);
    return ret;
}
```

n	FFFFFFFF	FFFFFFFF	62178293	A721B267
k	A928F516	22354820	AB2C6231	1A736227

bn_mul_mont is a fast, ASM optimized function

Another flaw

```
int bn_mul_mont_fixed_top(BIGNUM *r, const BIGNUM *a, const BIGNUM
    *b, BN_MONT_CTX *mont, BN_CTX *ctx)
{
    BIGNUM *tmp;
    int ret = 0;
    int num = mont->N.top;

    if (num > 1 && a->top == num && b->top == num) {
        if (bn_wexpand(r, num) == NULL)
            return 0;
        if (bn_mul_mont(r->d, a->d, b->d, mont->N.d, mont->n0, num)
    ) {
            r->neg = a->neg ^ b->neg;
            r->top = num;
            r->flags |= BN_FLG_FIXED_TOP;
            return 1;
        }
    }
    [...]
    if (a == b) {
        if (!bn_sqr_fixed_top(tmp, a, ctx))
            goto err;
    } else {
        if (!bn_mul_fixed_top(tmp, a, b, ctx))
            goto err;
    }
    /* reduce from aRR to aR */
    if (!bn_from_montgomery_word(r, tmp, mont))
        goto err;
    ret = 1;
 err:
    BN_CTX_end(ctx);
    return ret;
}
```

n	FFFFFFFF	FFFFFFFF	62178293	A721B267
k	00000000	22354820	AB2C6231	1A736227

bn_mul_fixed_top is a slow, high-level function

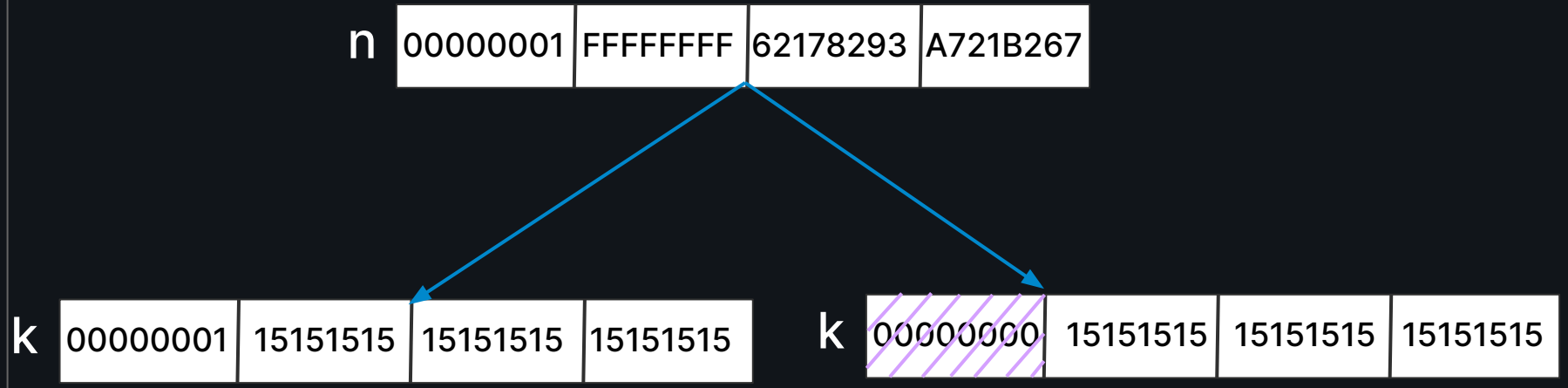
Timing attack: how to exploit?

1) Choose a clever n that can easily trigger the issue

n	00000001	FFFFFFFF	62178293	A721B267

Timing attack: how to exploit?

1) Choose a clever n that can easily trigger the issue



Timing attack: how to exploit?

1) Choose a clever n that can easily trigger the issue

n	00000001	FFFFFFFF	62178293	A721B267

2) Measure the execution times of several signatures

fastest

k	00000001	15151515	15151515	15151515

slowest

k	00000000	15151515	15151515	15151515

Timing attack: how to exploit?

1) Choose a clever n that can easily trigger the issue

n	00000001	FFFFFFFF	62178293	A721B267

2) Measure the execution times of several signatures

fastest

k	00000001	15151515	15151515	15151515

slowest

k	00000000	15151515	15151515	15151515

3) Recover the key from several nonces length (LLL) (See notebook!)

Vulnerable curves

Parameter n comes from an elliptic curve
OpenSSL allows the usage of many standard curves

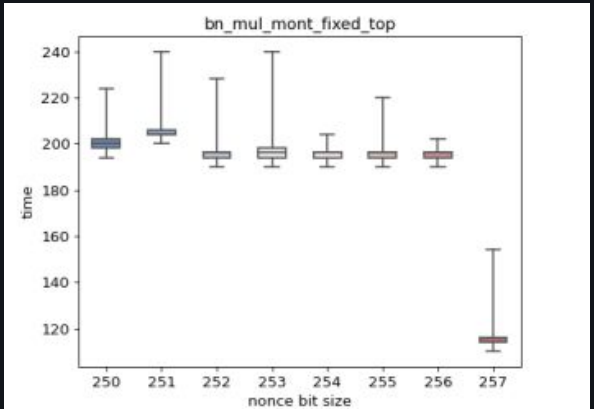
name	type	bias	name	type	bias
secp521r1	shorter	2^{-9}	c2pnb208w1	longer	2^{-4}
sect131r1	shorter	2^{-2}	c2pnb272w1	longer	2^{-8}
sect131r2	shorter	2^{-2}	c2pnb304w1	longer	2^{-4}
sect163k1	shorter	2^{-2}	c2pnb368w1	longer	2^{-8}
sect163r1	shorter	2^{-2}	c2tnb431r1	shorter	2^{-2}
sect163r2	shorter	2^{-2}	wap-wsg-id-ecid-wtls1	shorter	2^{-8}
sect233k1	shorter	2^{-7}	wap-wsg-idm-ecid-wtls3	shorter	2^{-2}
sect233r1	shorter	2^{-8}	wap-wsg-idm-ecid-wtls5	shorter	2^{-2}
sect409k1	shorter	2^{-7}	wap-wsg-idm-ecid-wtls10	shorter	2^{-7}
			wap-wsg-idm-ecid-wtls11	shorter	2^{-8}

Timing attack: practical setup

- OpenSSL 1.1.1k on a Raspberry Pi 4
- counting clock cycles using rdtsc
- choose a custom n of 257 bits

Timing attack: practical setup

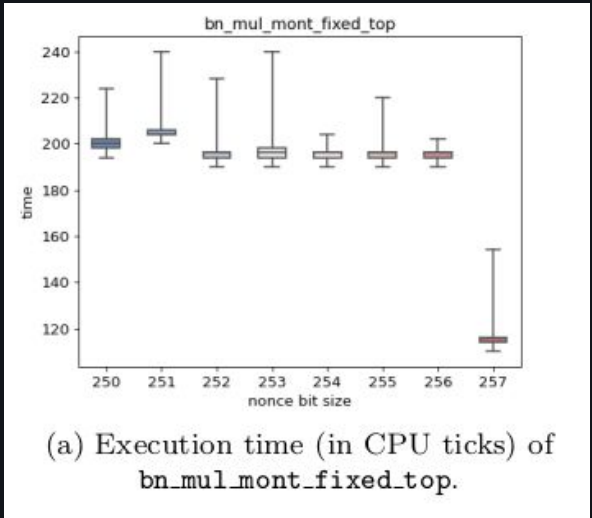
- OpenSSL 1.1.1k on a Raspberry Pi 4
- counting clock cycles using rdtsc
- choose a custom n of 257 bits



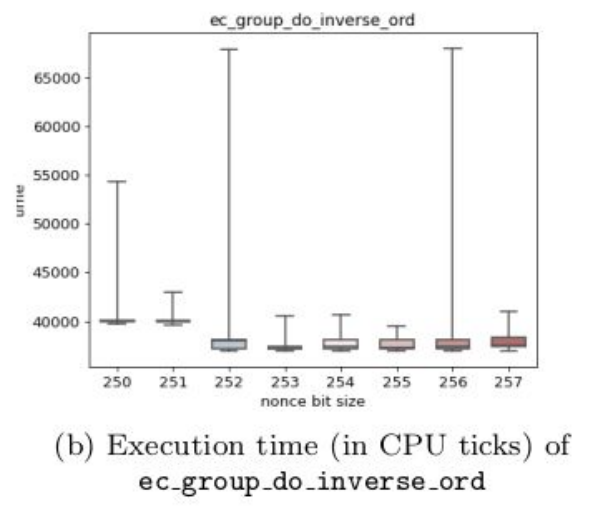
(a) Execution time (in CPU ticks) of `bn_mul_mont_fixed_top`.

Timing attack: practical setup

- OpenSSL 1.1.1k on a Raspberry Pi 4
- counting clock cycles using rdtsc
- choose a custom n of 257 bits



(a) Execution time (in CPU ticks) of `bn_mul_mont_fixed_top`.



(b) Execution time (in CPU ticks) of `ec_group_do_inverse_ord`

Timing results

Feasible:

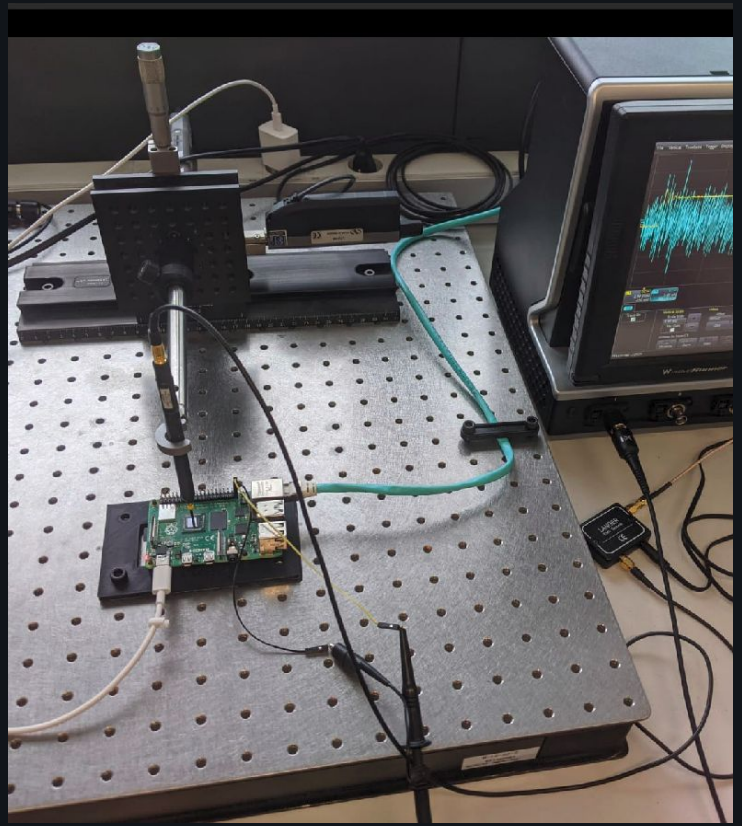
- If the attacker is able to clearly point out the beginning and the end of bn_mul_mont_fixed_top
- In a SGX-enclave / cache attack setup

Very hard:

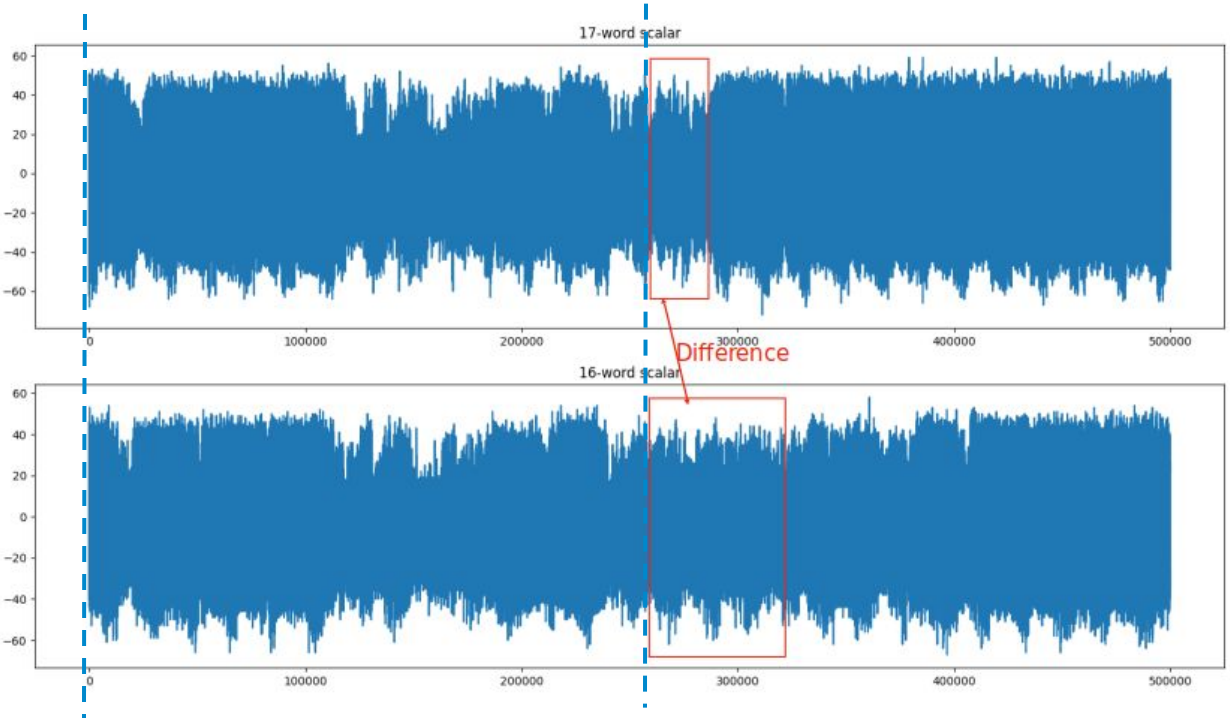
- The sensitive operation is too quick compared to the rest of the code ⇒ big noise

SEMA setup

- Measure the EM signal from the Raspberry during the computation
- EM probe
- Scope sampling 1GS/s
- Launch signature from the openssl command line



SEMA results



SEMA results

- Easy to detect the pattern, on the fly or offline
- Allows certain recovery of the nonce length
- Allows key recovery in practical time (LLL)
  (See notebook!)

How to fix this?

- The simplest way is to force the nonce to always be coded on the maximum number of words
- This implies many modifications in the bignum library
- This change was made eg. in BoringSSL

Takeaways

- If remote timings are not possible, a stronger attacker might find another way

- Please be extra-careful of your end usecase and the threat models when picking your lib