Timing Attacks against RSA

(DSP [1] - Project implementation)

Lorenzo Palloni

University of Florence

lorenzo.palloni@stud.unifi.it

January 24, 2022

[1] Data Security and Privacy

Introduction

- *What are we going to talk about?*
  - Some implementations related to the exam project.
- *What is the project about?*
  - An overview of two major timing attacks:
    - $\rightarrow$ Kocher's timing attack (1996) [1];
    - $\rightarrow$ Brumley and Boneh's timing attack (2005) [2].

Main source files

- kocher_main.py
  - → Kocher's timing attack;
- brumley_and_boneh_main.py
  - → Brumley and Boneh's timing attack;
- utilities.py
  - → Random prime number generator from scratch;
  - → RSA (that comes easily with the previous point);
  - → other utility functions.

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;

utilities.py

1. gen_rsa_factors(half_k: int $=$ 8, seed: int $=$ None) $\rightarrow$ Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int $=$ None, tries: int $=$ 3) $\rightarrow$ int;
3. gen_odd(low: int, high: int, seed: int $=$ None) $\rightarrow$ int;

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) $\rightarrow$ Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) $\rightarrow$ int;
3. gen_odd(low: int, high: int, seed: int = None) $\rightarrow$ int;
4. is_probably_prime(guess: int, tries: int = 4) $\rightarrow$ bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) $\rightarrow$ bool;

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) → bool;
6. gen_rabin_sequence(n: int, x: int, m: int, r: int) → List[int];

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) → bool;
6. gen_rabin_sequence(n: int, x: int, m: int, r: int) → List[int];
7. get_m_and_r(n: int) → Tuple[int, int];

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) → bool;
6. gen_rabin_sequence(n: int, x: int, m: int, r: int) → List[int];
7. get_m_and_r(n: int) → Tuple[int, int];
8. check_quadratic_residuals(n: int, seq: List[int]) → bool;

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) → bool;
6. gen_rabin_sequence(n: int, x: int, m: int, r: int) → List[int];
7. get_m_and_r(n: int) → Tuple[int, int];
8. check_quadratic_residuals(n: int, seq: List[int]) → bool;
9. mod_exp(a: int, m: int, n: int) → int;

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) → bool;
6. gen_rabin_sequence(n: int, x: int, m: int, r: int) → List[int];
7. get_m_and_r(n: int) → Tuple[int, int];
8. check_quadratic_residuals(n: int, seq: List[int]) → bool;
9. mod_exp(a: int, m: int, n: int) → int;
10. binarize(a: int) → List[int];

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) → bool;
6. gen_rabin_sequence(n: int, x: int, m: int, r: int) → List[int];
7. get_m_and_r(n: int) → Tuple[int, int];
8. check_quadratic_residuals(n: int, seq: List[int]) → bool;
9. mod_exp(a: int, m: int, n: int) → int;
10. binarize(a: int) → List[int];
11. binarize_inverse(a: List[int]) → int;

utilities.py

1. gen_rsa_factors(half_k: int = 8, seed: int = None) → Tuple[int, int];
2. gen_prime(low: int, high: int, seed: int = None, tries: int = 3) → int;
3. gen_odd(low: int, high: int, seed: int = None) → int;
4. is_probably_prime(guess: int, tries: int = 4) → bool;
5. rabin_test(n: int, x: int = None, verbose: bool = False) → bool;
6. gen_rabin_sequence(n: int, x: int, m: int, r: int) → List[int];
7. get_m_and_r(n: int) → Tuple[int, int];
8. check_quadratic_residuals(n: int, seq: List[int]) → bool;
9. mod_exp(a: int, m: int, n: int) → int;
10. binarize(a: int) → List[int];
11. binarize_inverse(a: List[int]) → int;
12. gcd(a: int, b: int) → Tuple[int, int].

kocher_main.py

- Kocher's timing attack;
- devices simulated with TimingAttackModule.py [2];
- dynamic number of ciphertexts for each iteration (i.e. for each bit).

Output example:

[2] Professor Michele Boreale provided it.

kocher_main.py

- Kocher's timing attack;
- devices simulated with TimingAttackModule.py [2];
- dynamic number of ciphertexts for each iteration (i.e. for each bit).

Output example:

```
➜  src git:(master) ✗ head kocher_output.txt && echo "\n...\n" && tail kocher_output.txt
# bit: 2/64     Number of ciphertexts: 10000    Ratio of recovered bits: 0.484375
# bit: 3/64     Number of ciphertexts: 9458     Ratio of recovered bits: 0.5
# bit: 4/64     Number of ciphertexts: 8945     Ratio of recovered bits: 0.5
# bit: 5/64     Number of ciphertexts: 8460     Ratio of recovered bits: 0.5
# bit: 6/64     Number of ciphertexts: 8002     Ratio of recovered bits: 0.5
# bit: 7/64     Number of ciphertexts: 7568     Ratio of recovered bits: 0.5
# bit: 8/64     Number of ciphertexts: 7158     Ratio of recovered bits: 0.515625
# bit: 9/64     Number of ciphertexts: 6770     Ratio of recovered bits: 0.515625
# bit: 10/64    Number of ciphertexts: 6404     Ratio of recovered bits: 0.53125
# bit: 11/64    Number of ciphertexts: 6057     Ratio of recovered bits: 0.546875

...

# bit: 57/64    Number of ciphertexts: 467      Ratio of recovered bits: 0.9375
# bit: 58/64    Number of ciphertexts: 441      Ratio of recovered bits: 0.9375
# bit: 59/64    Number of ciphertexts: 417      Ratio of recovered bits: 0.953125
# bit: 60/64    Number of ciphertexts: 395      Ratio of recovered bits: 0.96875
# bit: 61/64    Number of ciphertexts: 373      Ratio of recovered bits: 0.96875
# bit: 62/64    Number of ciphertexts: 353      Ratio of recovered bits: 0.984375
# bit: 63/64    Number of ciphertexts: 334      Ratio of recovered bits: 0.984375
# bit: 64/64    Number of ciphertexts: 316      Ratio of recovered bits: 1.0
100% of key bits recovered.
```

[2] Professor Michele Boreale provided it.

brumley_and_boneh_main.py

- Brumley and Boneh's timing attack;
- Two classes implemented:
  - class *Device*
  - class *Attacker*

brumley_and_boneh_main.py

- class *Device*

brumley_and_boneh_main.py

- class *Device*
  - __init__(

```
self,
num_bits: int = 16,
seed: int = None,
blinding: bool = False
```

  );

- class *Device*
  - __init__(
    self,
    num_bits: int = 16,
    seed: int = None,
    blinding: bool = False
    );
  - gen_montgomery_coefficient(self) → int;

brumley_and_boneh_main.py

- class *Device*
  - __init__(

    self,

    num_bits: int = 16,

    seed: int = None,

    blinding: bool = False

    );
  - gen_montgomery_coefficient(self) → int;
  - get_modulus(self) → int;

brumley_and_boneh_main.py

- class *Device*
  - __init__(

        self,
        num_bits: int = 16,
        seed: int = None,
        blinding: bool = False

    );
  - gen_montgomery_coefficient(self) → int;
  - get_modulus(self) → int;
  - run(self, u: int) → float;

brumley_and_boneh_main.py

- class *Device*
  - __init__(

    self,
    num_bits: int = 16,
    seed: int = None,
    blinding: bool = False

    );
  - gen_montgomery_coefficient(self) → int;
  - get_modulus(self) → int;
  - run(self, u: int) → float;
  - _decryption(self, u: int) → float;

- class *Device*
  - __init__(
    ```
    self,
    num_bits: int = 16,
    seed: int = None,
    blinding: bool = False
    ```
    );
  - gen_montgomery_coefficient(self) → int;
  - get_modulus(self) → int;
  - run(self, u: int) → float;
  - _decryption(self, u: int) → float;
  - _get_factors(self) → Tuple[int, int];

brumley_and_boneh_main.py

- class *Attacker*

brumley_and_boneh_main.py

- class *Attacker*
  - __init__(

```
self,
device: Device,
num_bits_per_factor: int = None,
modulus: int = None,
montgomery_coefficient: int = None,
```

    );

- class *Attacker*
  - __init__(

    self,
    device: Device,
    num_bits_per_factor: int = None,
    modulus: int = None,
    montgomery_coefficient: int = None,

    );
  - guess(self, threshold: float = 4e-4) → int;

brumley_and_boneh_main.py

- class *Attacker*
  - __init__(

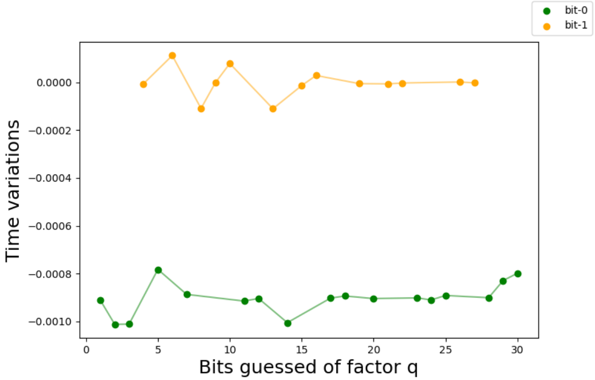
        self,
        device: Device,
        num_bits_per_factor: int = None,
        modulus: int = None,
        montgomery_coefficient: int = None,

    );
  - guess(self, threshold: float = 4e-4) → int;
  - plot_last_guess(self, savefig_path=None, figsize=None).

brumley_and_boneh_main.py

Code snippet

1. device = Device(num_bits=64, seed=42, blinding=False)
2. attacker = Attacker(device)
3. g = attacker.guess()
4. attacker.plot_last_guess()

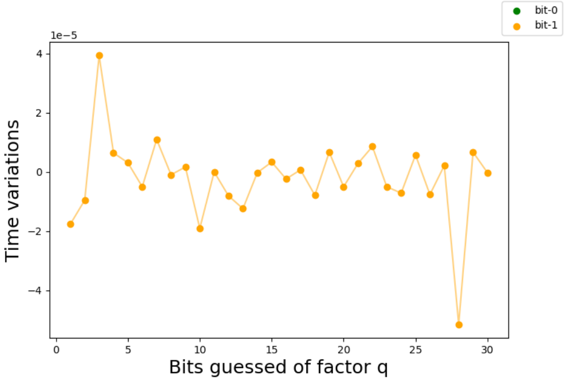


- *What is in the guess* $\mathbf{g}$ *of this example?*
  - $\rightarrow$ 2330302001
- *...and the actual factor* $\mathbf{q}$*?*
  - $\rightarrow$ 2330302003
- *How does the guess* **binarize(g)** *look like?*
  - $\rightarrow$ $[1, 0, 0, 1, 0, \dots, 1, 0, 0, \mathbf{0}, \mathbf{1}]$
- *...and what about* **binarize(q)***?*
  - $\rightarrow$ $[1, 0, 0, 1, 0, \dots, 1, 0, 0, \mathbf{1}, \mathbf{1}]$

brumley_and_boneh_main.py

Code snippet

1. device = Device(num_bits=64, seed=42, blinding=**True**)
2. attacker = Attacker(device)
3. g = attacker.guess()
4. attacker.plot_last_guess()



Device._decryption(self, u: int) $\rightarrow$ float:

1. convert the input u in its Montgomery form $\rightarrow g$;
2. initializes $t_q := 0$ and $t_p := 0$;
3. if $g < self.q$, then:
   - $t_q = t_q + 1000$ (many Montgomery reductions);
   - $t_q = t_q + 100$ (normal multiplication routine);

   otherwise ($g \geq self.q$):
   - $t_q = t_q + 10$ (few Montgomery reductions);
   - $t_q = t_q + 10$ (Karatsuba multiplication routine);
4. repeat step 3. with $self.p$ (updating t_p);
5. $time.sleep(\frac{\mathcal{N}(t_q+t_p,\ 5)}{1e6})$.

Do you have any questions?

References

Kocher, P.C., 1996, August. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Annual International Cryptology Conference (pp. 104-113). Springer, Berlin, Heidelberg.

Brumley, D. and Boneh, D., 2005. Remote timing attacks are practical. Computer Networks, 48(5), pp.701-716.