PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild

Daniel De Almeida Braga

Pierre-Alain Fouque

Mohamed Sabt

CCS’21 - November 2021

Context and Motivations

A Few Words About PAKEs

What to expect from a PAKE, starting from a password:

- Authentication
- End up with strong key
- Resist to (offline) dictionary attack

Lots of different PAKEs (two main families: balanced - asymmetric).

Why Looking at PAKEs?

Recent interest:

- Wide Deployment of Dragonfly in WPA3
- CFRG competition for new standard (OPAQUE and CPace)

Why Looking at PAKEs?

Recent interest:

- Wide Deployment of Dragonfly in WPA3
- CFRG competition for new standard (OPAQUE and CPace)

Practical security considerations

- Dragonfly and WPA3: Dragonblood[1] and attack refinement[2]
- Partitioning Oracle Attack[3] applied to some OPAQUE implementations

Case study: Secure Remote Password

[1] M.Vanhoef and E.Ronen *Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd*. In IEEE S&P. 2020

[2] D.Braga et al. *Dragonblood Is Still Leaking: Practical Cache-based Side-Channel in the Wild*. In ACSAC. 2020

[3] J.Len et al. *Partitioning Oracle Attack*. In USENIX Security. 2021

What about SRP?

Available for a long time => de facto standard for more than 20 years

What about SRP implementations in the wild ?

What about SRP?

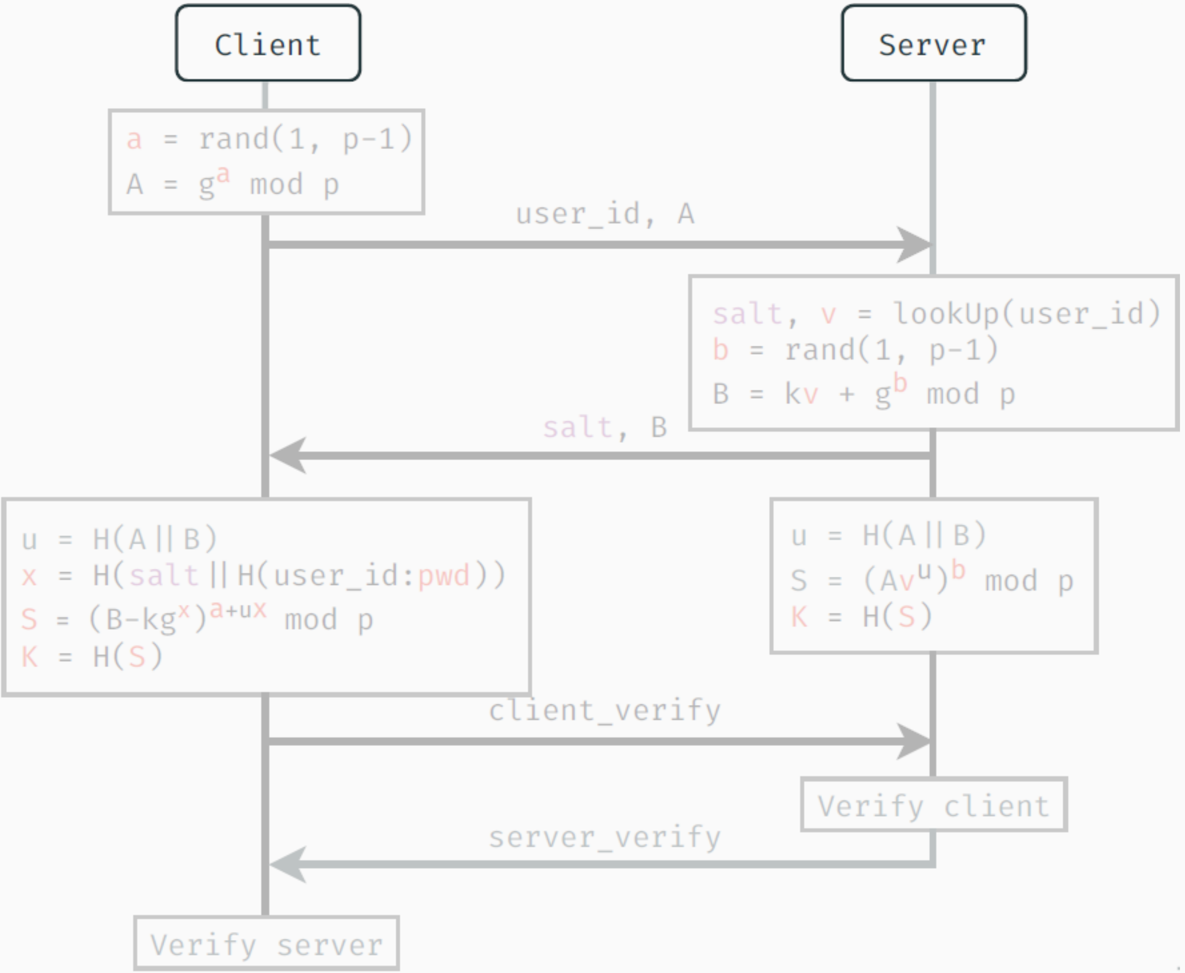
Available for a long time => de facto standard for more than 20 years

What about SRP implementations in the wild ?

- Recent work on SRP at ACNS[1]

[1] A.Russon Threat for the Secure Remote Password Protocol and a Leak in Apple’s Cryptographic Library. In ACNS. 2021

SRP Protocol Overview



Client

Server

a = rand(1, p-1)
$A = g^{a} \bmod p$

user_id, A

salt, v = lookUp(user_id)
b = rand(1, p-1)
$B = kv + g^{b} \bmod p$

salt, B

$u = H(A \| B)$
$x = H(salt \| H(user\_id:pwd))$
$S = (B - kg^{x})^{a+ux} \bmod p$
$K = H(S)$

$u = H(A \| B)$
$S = (Av^{u})^{b} \bmod p$
$K = H(S)$

client_verify

Verify client

server_verify

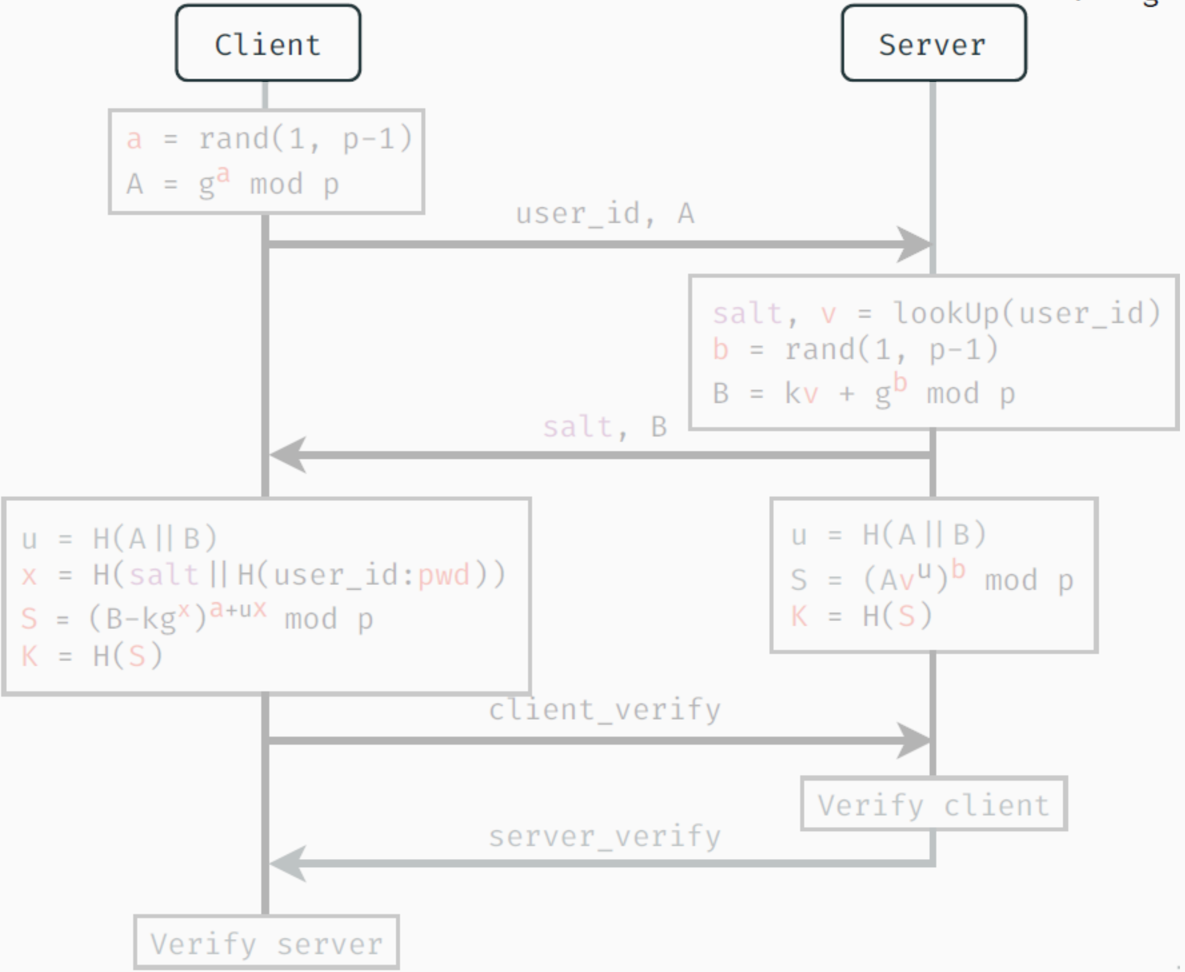
Verify server

SRP Protocol Overview

$x = H(salt, H(id{:}pwd))$

$v = g^x \bmod p$

pwd



Client

Server

$a = rand(1, p-1)$

$A = g^a \bmod p$

user_id, A

$salt, v = lookUp(user\_id)$

$b = rand(1, p-1)$

$B = kv + g^b \bmod p$

salt, B

$u = H(A || B)$

$x = H(salt || H(user\_id{:}pwd))$

$S = (B - kg^x)^{a+ux} \bmod p$

$K = H(S)$

$u = H(A || B)$

$S = (Av^u)^b \bmod p$

$K = H(S)$

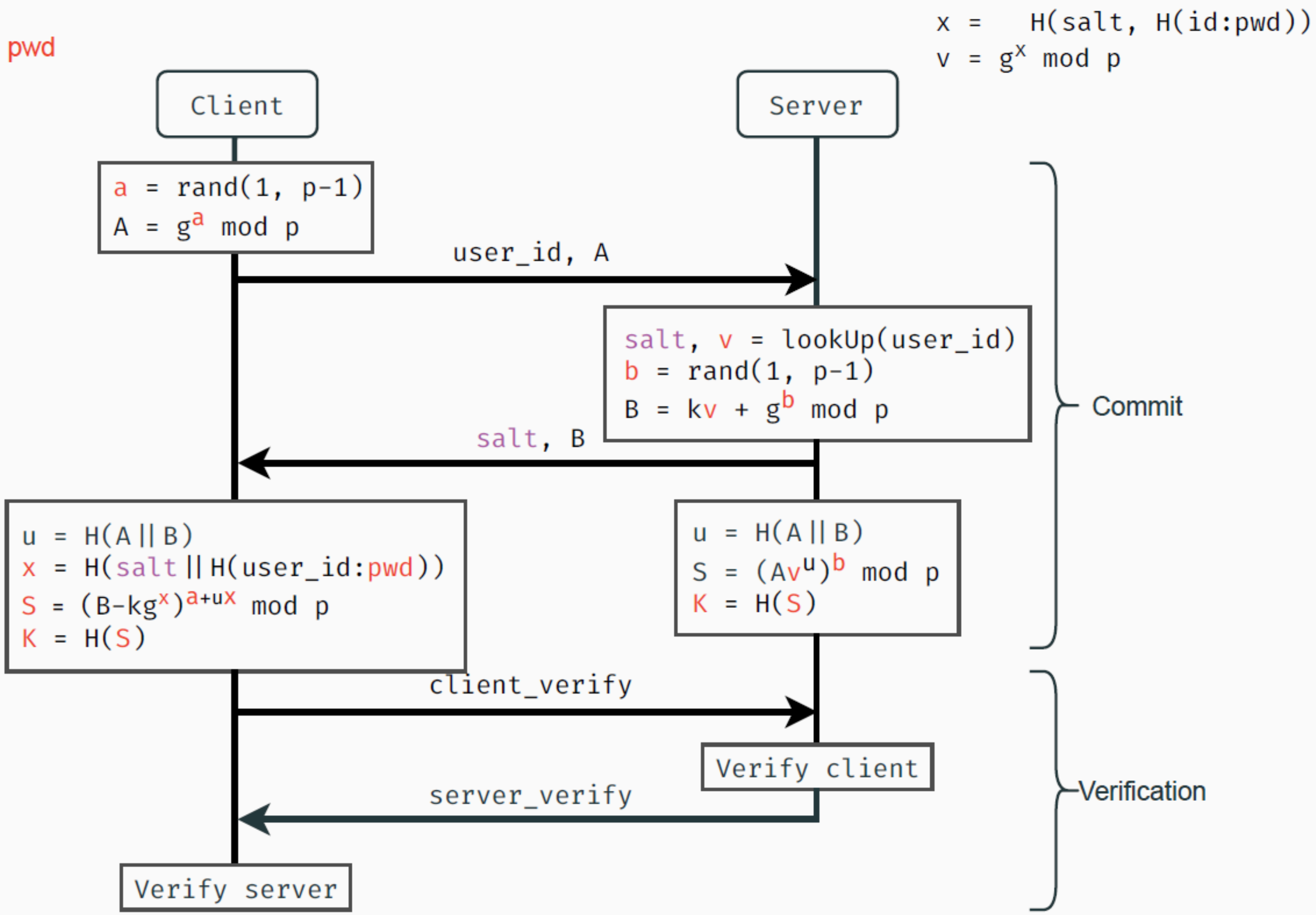
client_verify

Verify client

server_verify

Verify server

SRP Protocol Overview



pwd

x = H(salt, H(id:pwd))

$v = g^x \bmod p$

Client

Server

a = rand(1, p-1)

$A = g^a \bmod p$

user_id, A

salt, v = lookUp(user_id)

b = rand(1, p-1)

$B = kv + g^b \bmod p$

salt, B

u = H(A || B)

x = H(salt || H(user_id:pwd))

$S = (B - kg^x)^{a+ux} \bmod p$

K = H(S)

u = H(A || B)

$S = (Av^u)^b \bmod p$

K = H(S)

Commit

client_verify

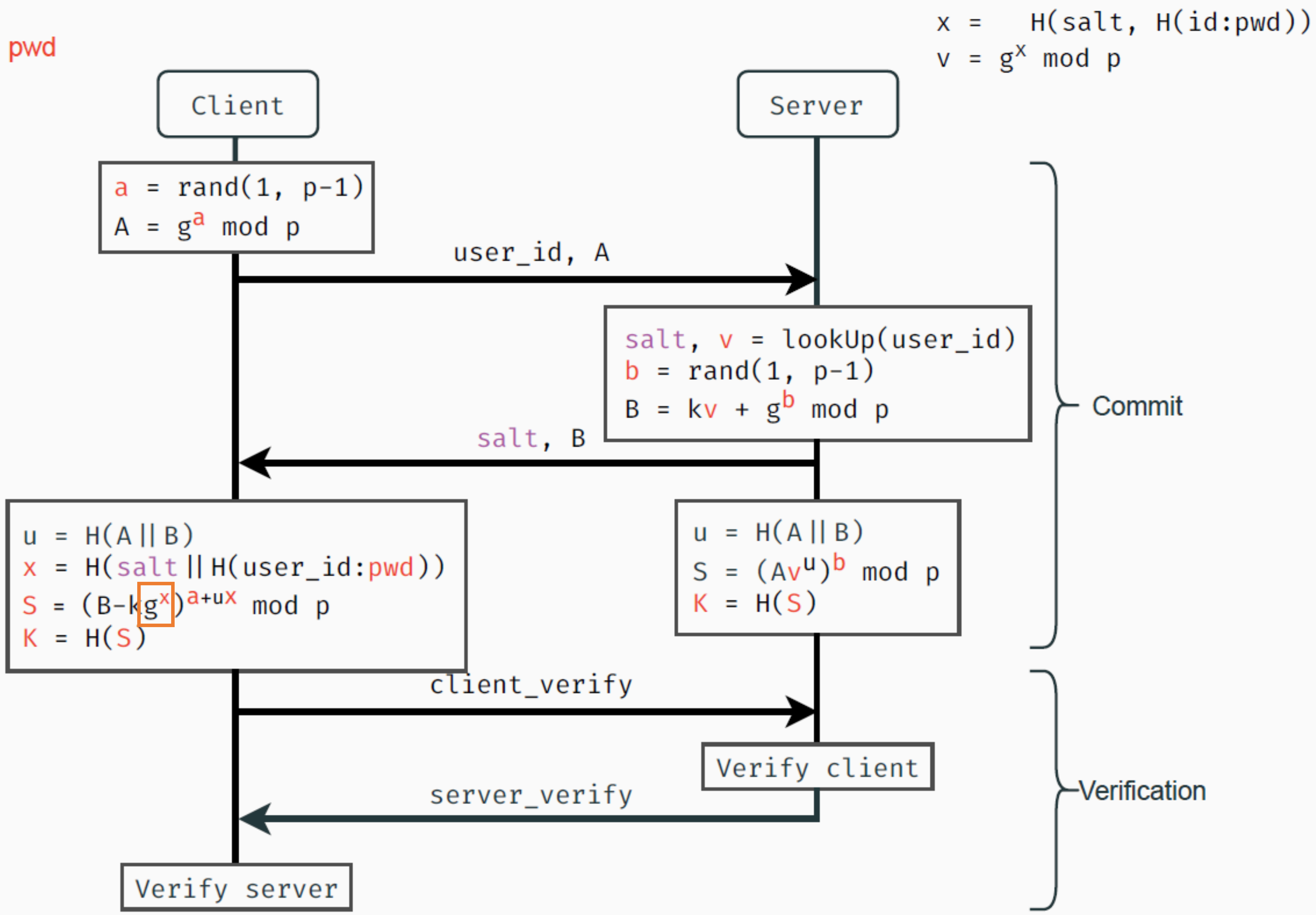
Verify client

server_verify

Verify server

Verification

SRP Protocol Overview



pwd

$x = H(salt, H(id:pwd))$

$v = g^{x} \bmod p$

Client

Server

$a = rand(1, p-1)$

$A = g^{a} \bmod p$

user_id, A

salt, v = lookUp(user_id)

$b = rand(1, p-1)$

$B = kv + g^{b} \bmod p$

salt, B

$u = H(A || B)$

$x = H(salt || H(user\_id:pwd))$

$S = (B-k\boxed{g^{x}})^{a+ux} \bmod p$

$K = H(S)$

$u = H(A || B)$

$S = (Av^{u})^{b} \bmod p$

$K = H(S)$

Commit

client_verify

Verify client

server_verify

Verify server

Verification

Contributions

Contributions

1. Study of various SRP implementations
2. Highlight a leakage in the root library used for big number arithmetic (OpenSSL)
3. Design PoCs of an offline dictionary attack recovering the password on impacted projects
4. Outline the importance of SCA, especially for PAKEs

Our Main Result

A cache-attack that lets us extract information

during OpenSSL the modular exponentiation

allowing to recover the password in a single measure.

Our Main Result

Flush+Reload[1] and PDA[2]

A cache-attack that lets us extract information

during OpenSSL the modular exponentiation

allowing to recover the password in a single measure.

[1] Y. Yarom et al. *Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. In USENIX Security Symposium. 2014

[2] T. Allan et al. *Amplifying side channels through performance degradation*. In ACSAC. 2016

Our Main Result

Flush+Reload[1] and PDA[2]

Weak exponentiation algorithm

A cache-attack that lets us extract information during OpenSSL the modular exponentiation allowing to recover the password in a single measure.

[1] Y. Yarom et al. *Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. In USENIX Security Symposium. 2014

[2] T. Allan et al. *Amplifying side channels through performance degradation*. In ACSAC. 2016

Our Main Result

Flush+Reload[1] and PDA[2]

Weak exponentiation algorithm

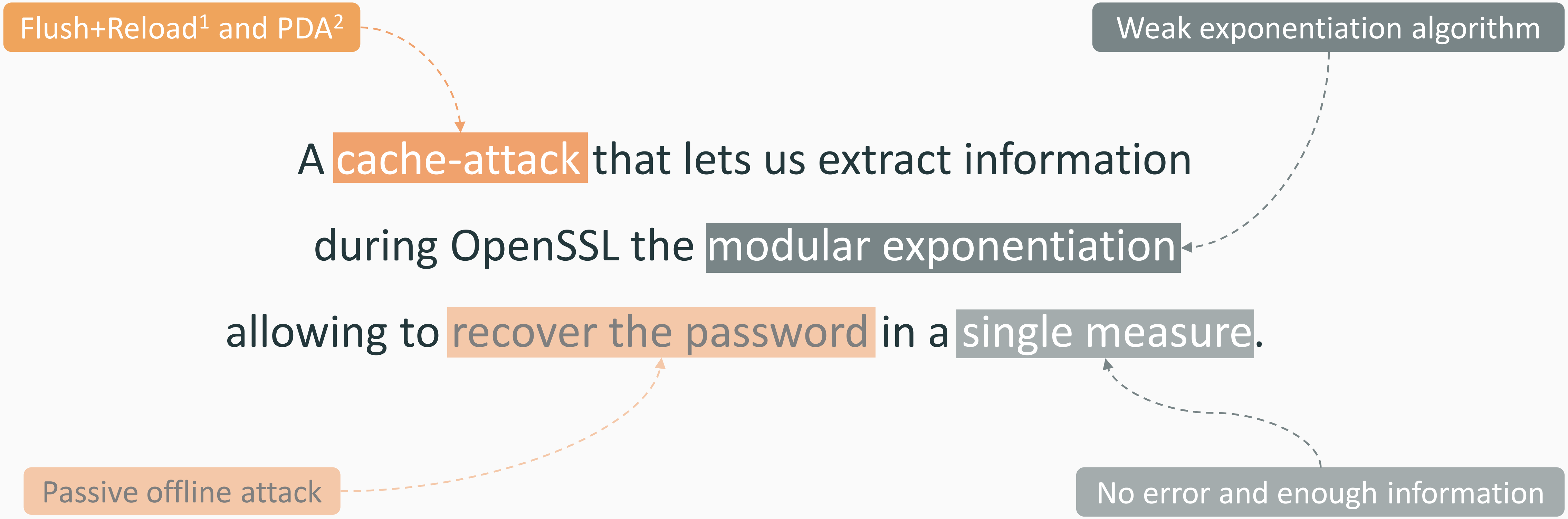
A cache-attack that lets us extract information during OpenSSL the modular exponentiation allowing to recover the password in a single measure.

Passive offline attack

[1] Y. Yarom et al. *Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. In USENIX Security Symposium. 2014

[2] T. Allan et al. *Amplifying side channels through performance degradation*. In ACSAC. 2016

Our Main Result



Flush+Reload[1] and PDA[2]

Weak exponentiation algorithm

A cache-attack that lets us extract information

during OpenSSL the modular exponentiation

allowing to recover the password in a single measure.

Passive offline attack

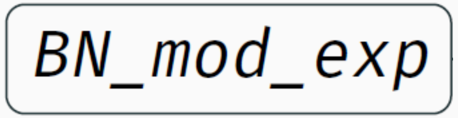
No error and enough information

[1] Y. Yarom et al. *Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. In USENIX Security Symposium. 2014

[2] T. Allan et al. *Amplifying side channels through performance degradation*. In ACSAC. 2016

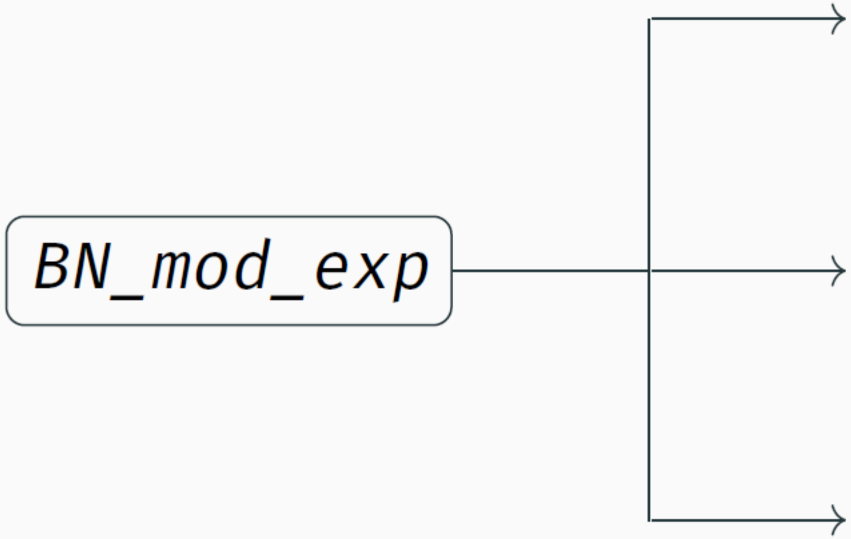
The Vulnerability

Modular exponentiation in OpenSSL

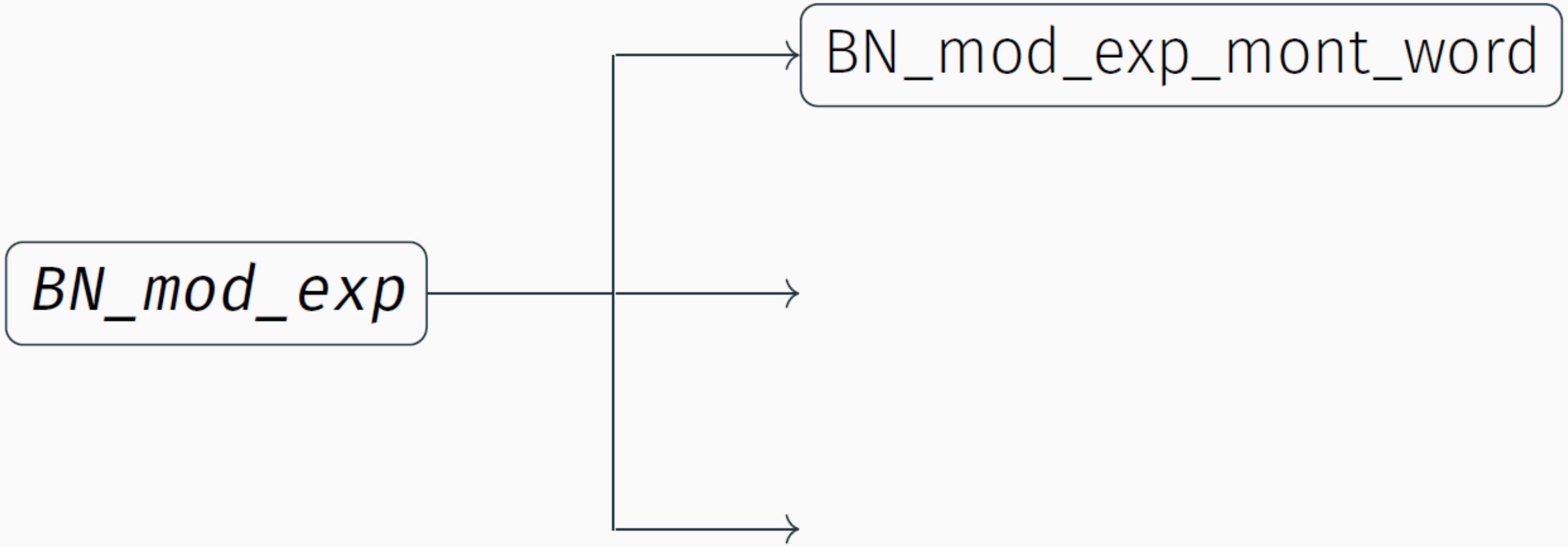


BN_mod_exp

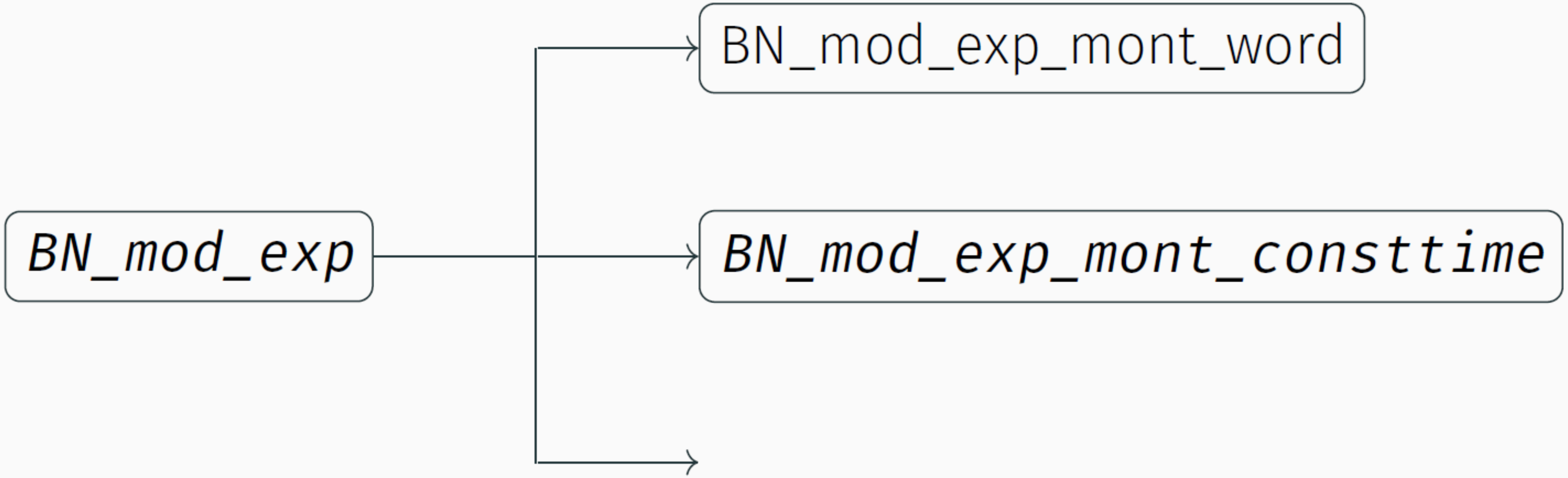
Modular exponentiation in OpenSSL



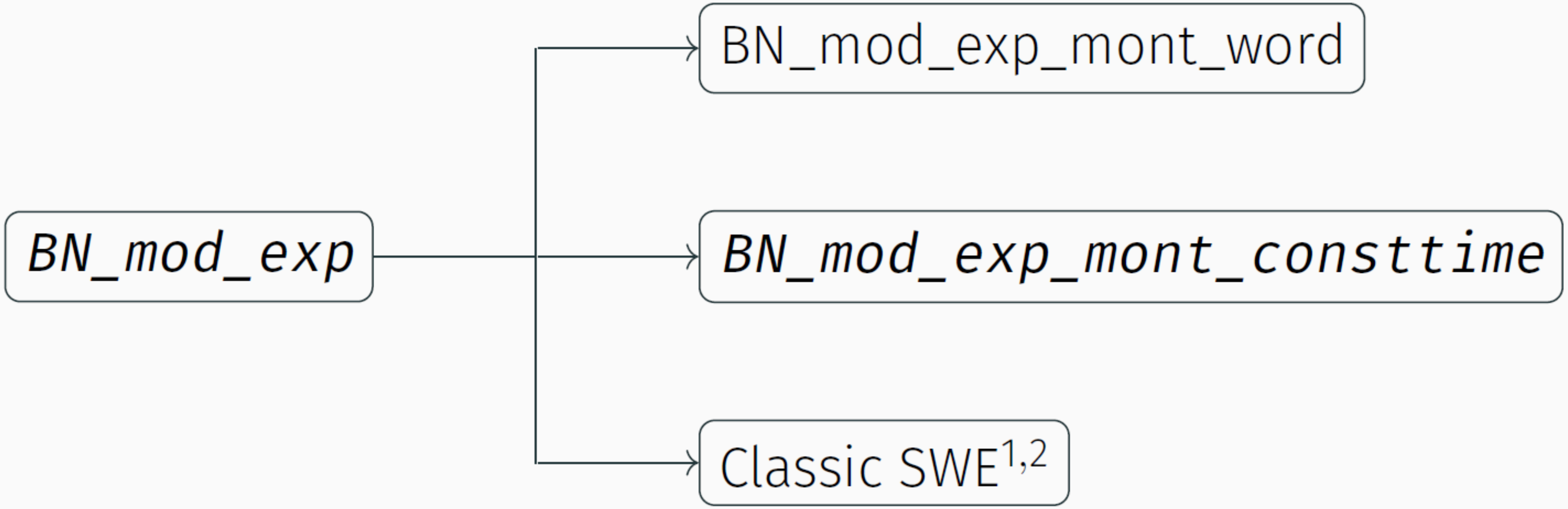
Modular exponentiation in OpenSSL



Modular exponentiation in OpenSSL



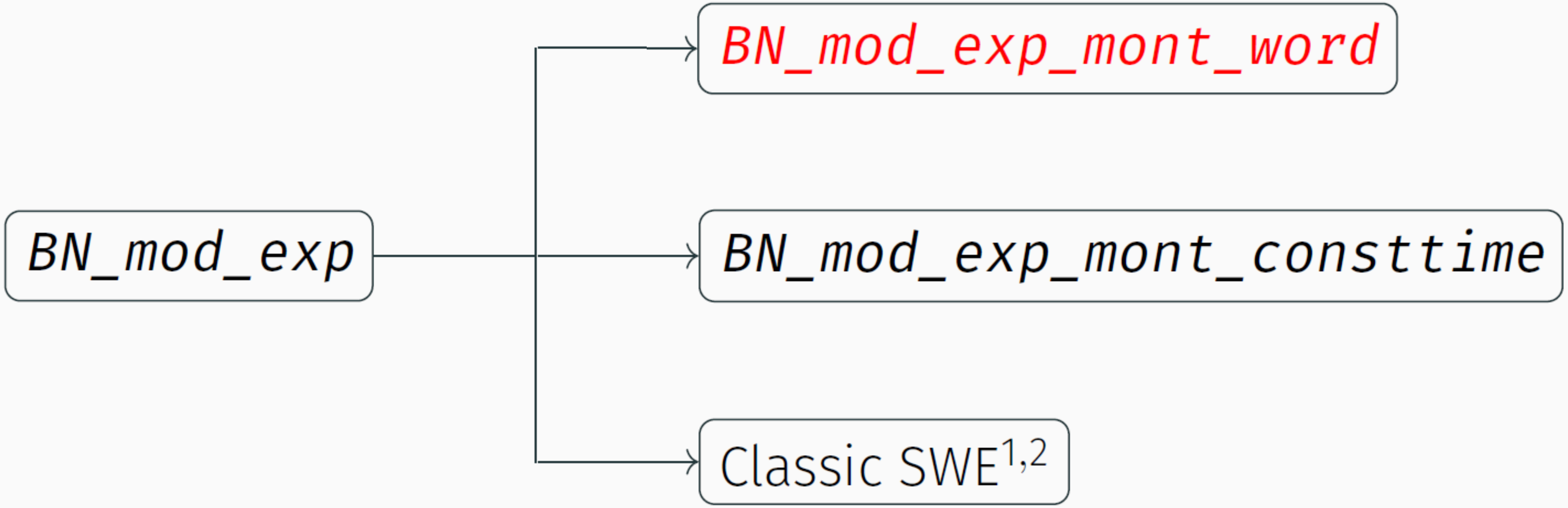
Modular exponentiation in OpenSSL



[1] C. Percival Cache missing for fun and profit. 2005

[2] C. Peraida Garia et al. Certified Side Channels. In USENIX Security. 2020

Modular exponentiation in OpenSSL



[1] C. Percival Cache missing for fun and profit. 2005

[2] C. Peraida Garia et al. Certified Side Channels. In USENIX Security. 2020

Optimized Square-and-Multiply

$$bin(e) = 1\ 1\ 0\ 1\ 0\ .\ .\ .$$

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)

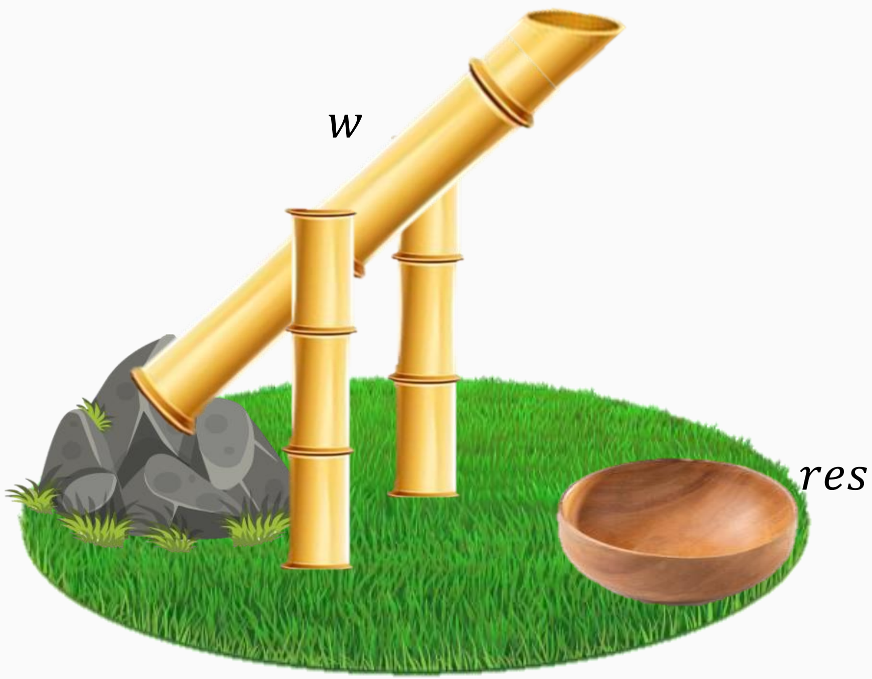
```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                     # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Optimized Square-and-Multiply

$bin(e) = 1\ 1\ 0\ 1\ 0\ . . .$

$res = g^{e} mod\ p$

w is a processor word (e.g. 64 bits)



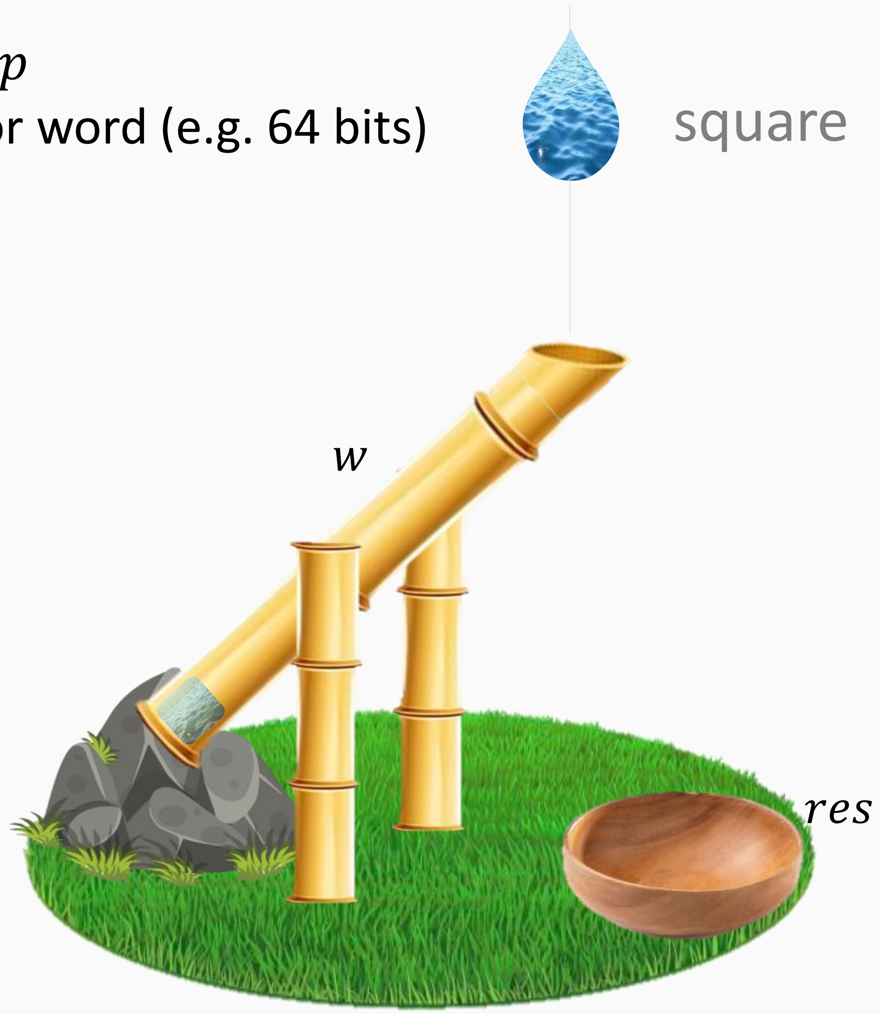
```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                      # uint64_t
    res = BN_to_mont_word(w)   # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Optimized Square-and-Multiply

$bin(e) =$ 1 1 0 1 0 . . .

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)



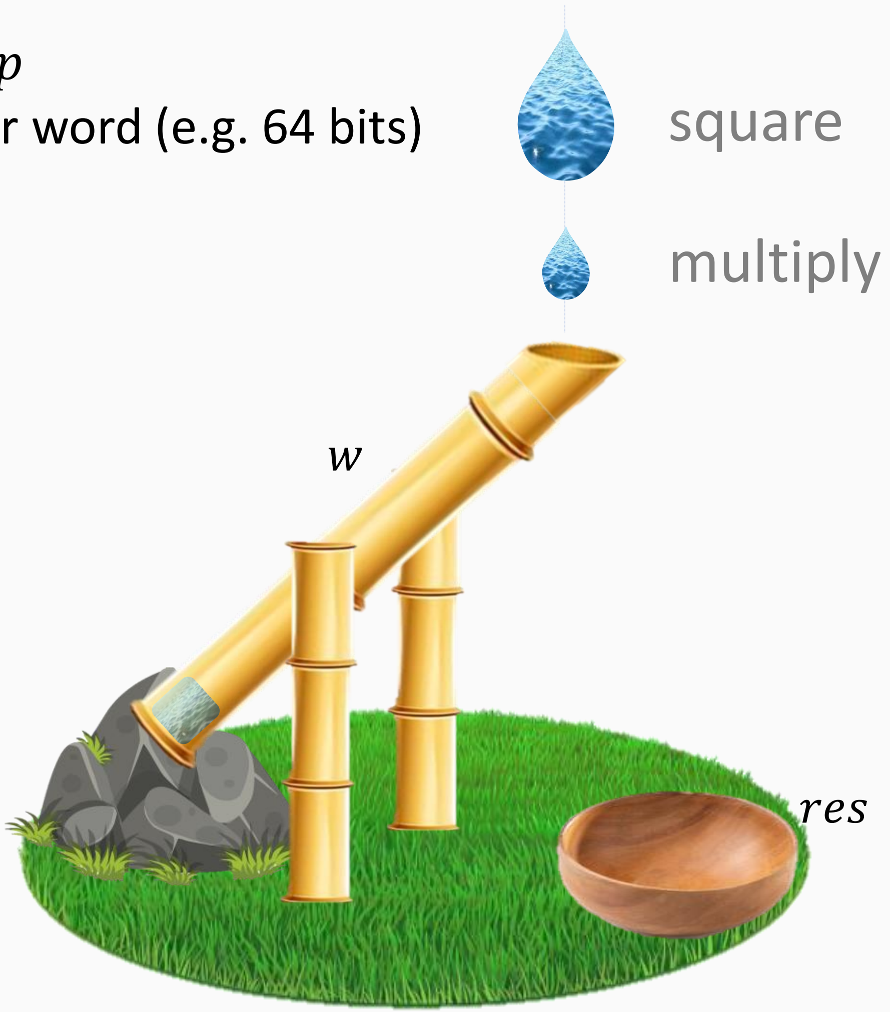
```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                     # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
    →   next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Optimized Square-and-Multiply

$bin(e) = 1\ 1\ 0\ 1\ 0\ .\ .\ .$

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)



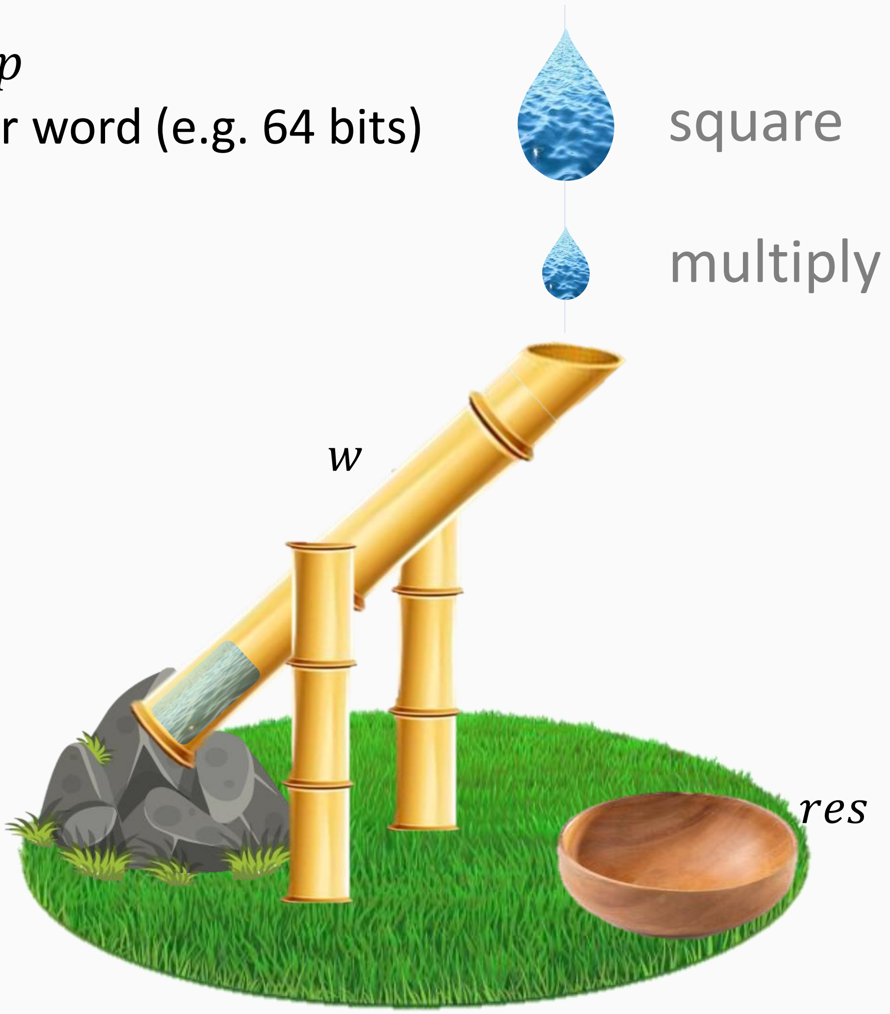
```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                      # uint64_t
    res = BN_to_mont_word(w)   # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
      ⟶    next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Optimized Square-and-Multiply

$bin(e) = 1\ 1\ 0\ 1\ 0\ .\ .\ .$

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)



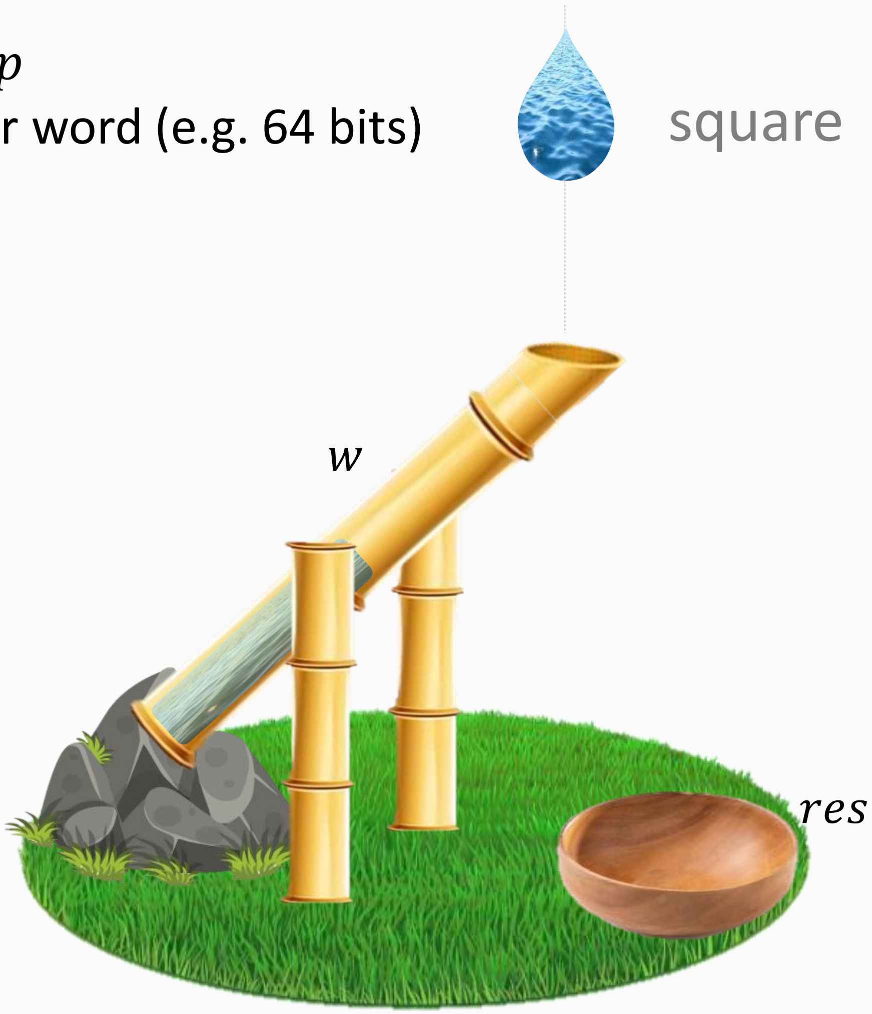
```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                         # uint64_t
    res = BN_to_mont_word(w)      # bigum
    for b in range(bitlen-2, 0, -1):
     ⟶  next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
         ⟶  next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Optimized Square-and-Multiply

$bin(e) = 1\ 1\ 0\ 1\ 0\ .\ .\ .$

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)



```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                          # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
  ⟶    next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Optimized Square-and-Multiply

$bin(e) =$ 1 1 0 1 0 . . .

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)

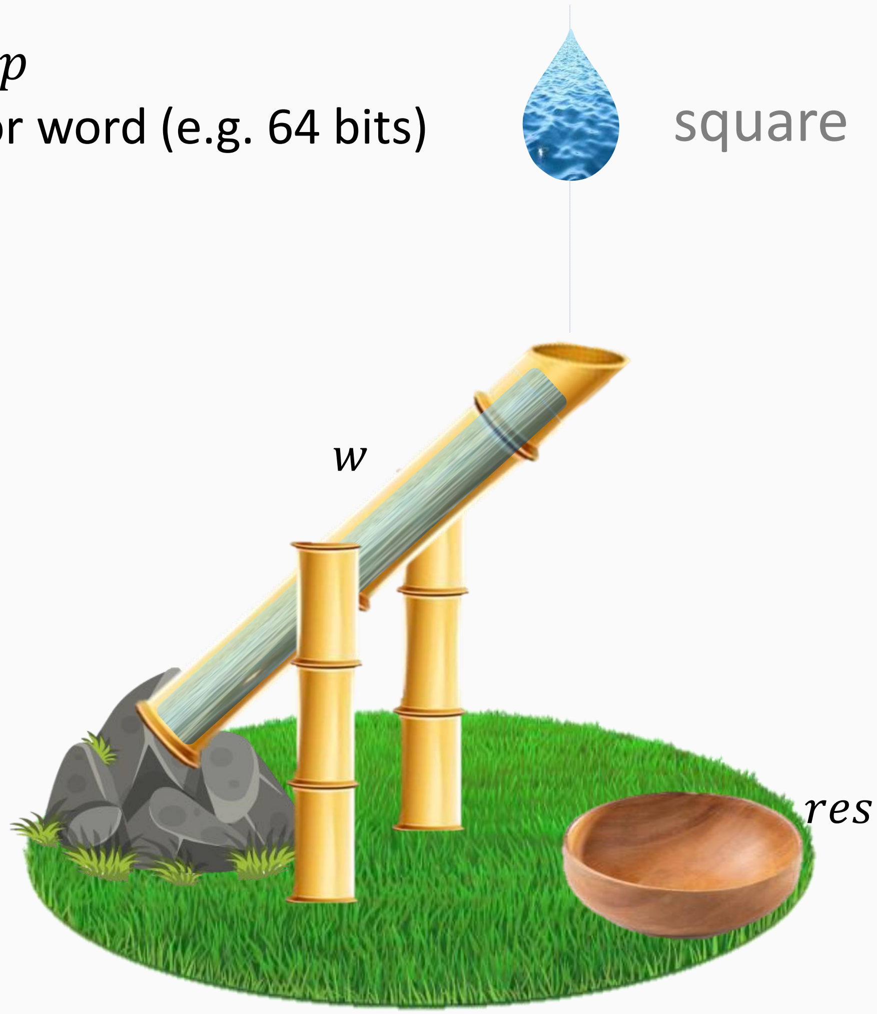
```
def BN_mod_exp_mot_word(g, w, p):
  …
  w = g                       # uint64_t
  res = BN_to_mont_word(w)    # bigum
  for b in range(bitlen-2, 0, -1):
  → next_w = w x w
    if next_w/w != w:
      res = BN_mod_mul(res, w, p)
      next_w = 1
    w = next_w
    res = BN_sqr(res)
    if BN_is_bit_set(x, b):
    → next_w = w x g
      if next_w/g != w:
        res = BN_mod_mul(res, w, p)
        next_w = g
      w = next_w
  …
```

Optimized Square-and-Multiply

$bin(e) =$ 1 1 0 1 0 . . .

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)



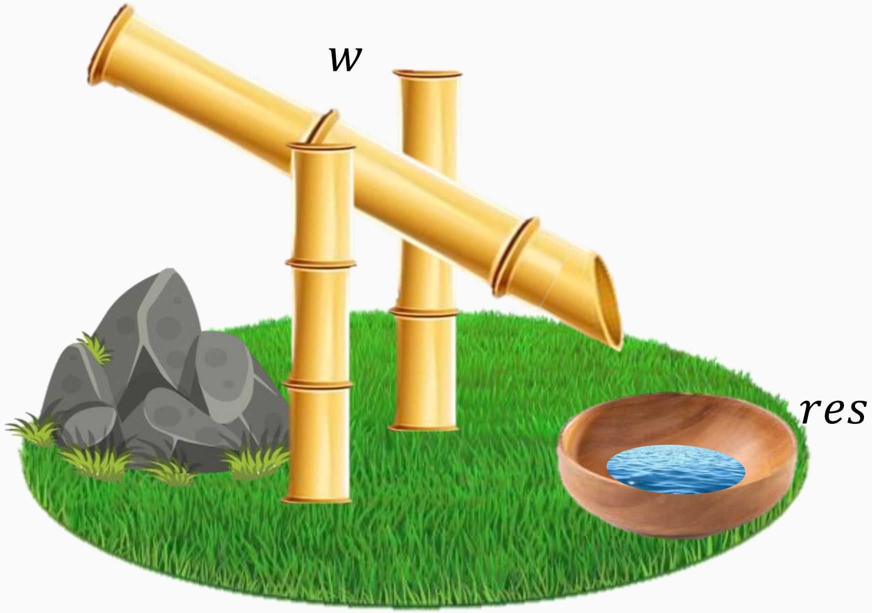
```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                     # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Optimized Square-and-Multiply

$bin(e) =$ 1 1 0 1 0 . . .

$res = g^e mod\ p$

w is a processor word (e.g. 64 bits)



```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                     # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Exploiting the Leakage

Trace Acquisition

```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                     # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
        res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Trace Acquisition

```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                     # uint64_t
    res = BN_to_mont_word(w)  # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
  🕵 ──→ res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Trace Acquisition

```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                         # uint64_t
    res = BN_to_mont_word(w)      # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
🕵 ──→  res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Trace Acquisition

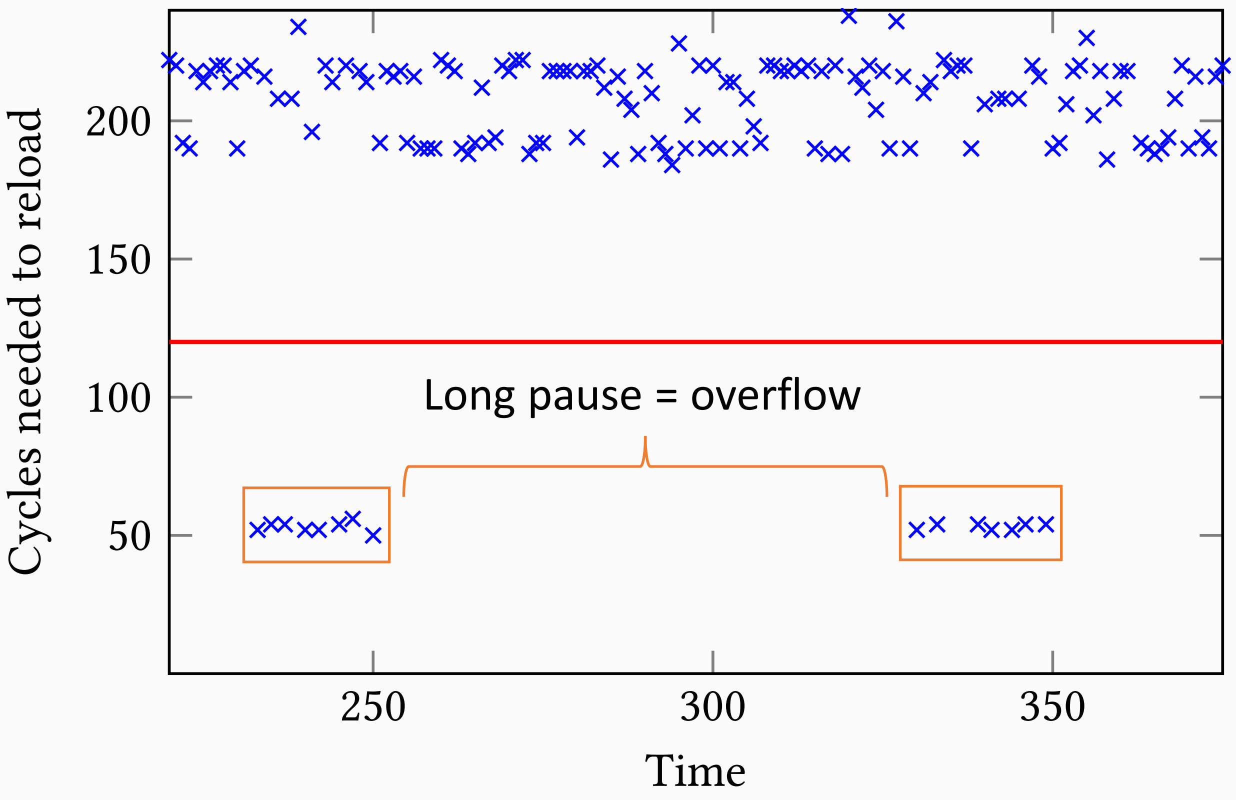
```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                         # uint64_t
    res = BN_to_mont_word(w)      # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
            res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
 🕵 →   res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

Trace Acquisition

```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                         # uint64_t
    res = BN_to_mont_word(w)      # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
🐌 ───────→ res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
🕵 ───→ res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

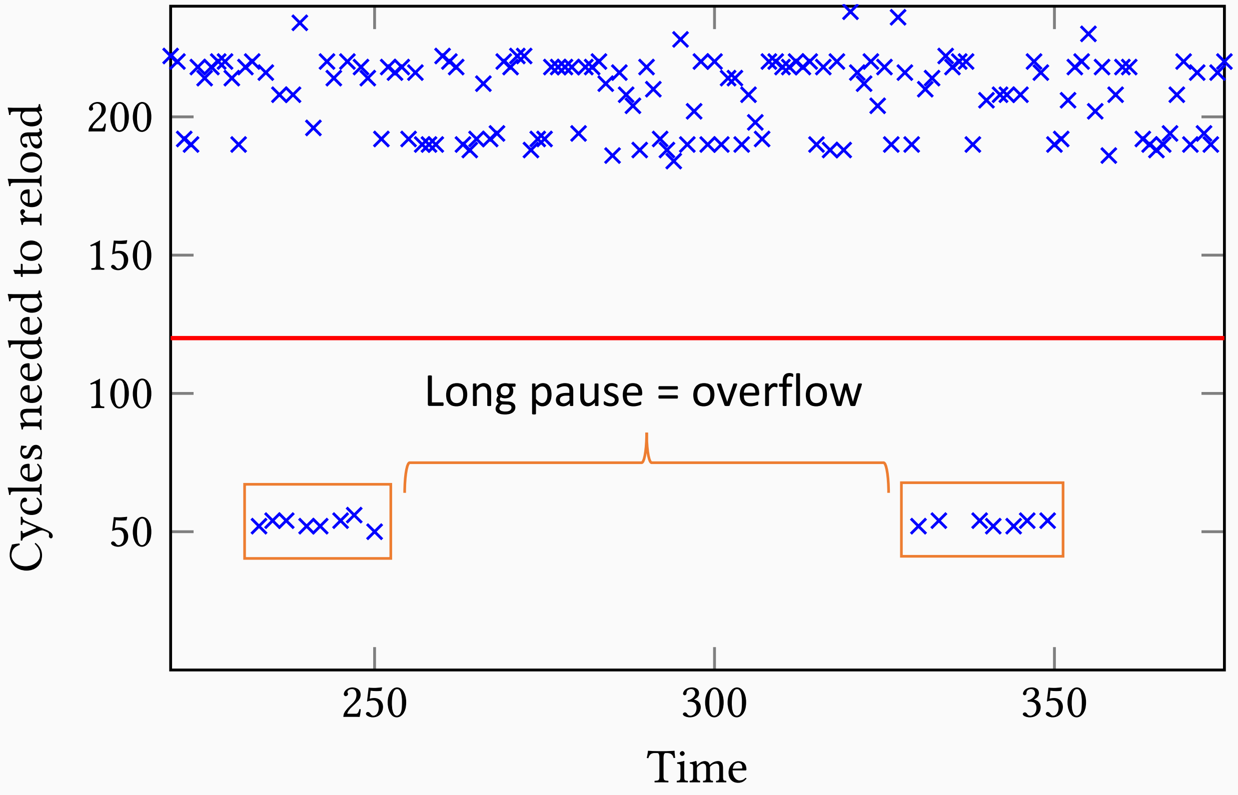
Trace Acquisition

```
def BN_mod_exp_mot_word(g, w, p):
    …
    w = g                         # uint64_t
    res = BN_to_mont_word(w)      # bigum
    for b in range(bitlen-2, 0, -1):
        next_w = w x w
        if next_w/w != w:
🐌 ───→    res = BN_mod_mul(res, w, p)
            next_w = 1
        w = next_w
🕵 ─→  res = BN_sqr(res)
        if BN_is_bit_set(x, b):
            next_w = w x g
            if next_w/g != w:
                res = BN_mod_mul(res, w, p)
                next_w = g
            w = next_w
    …
```

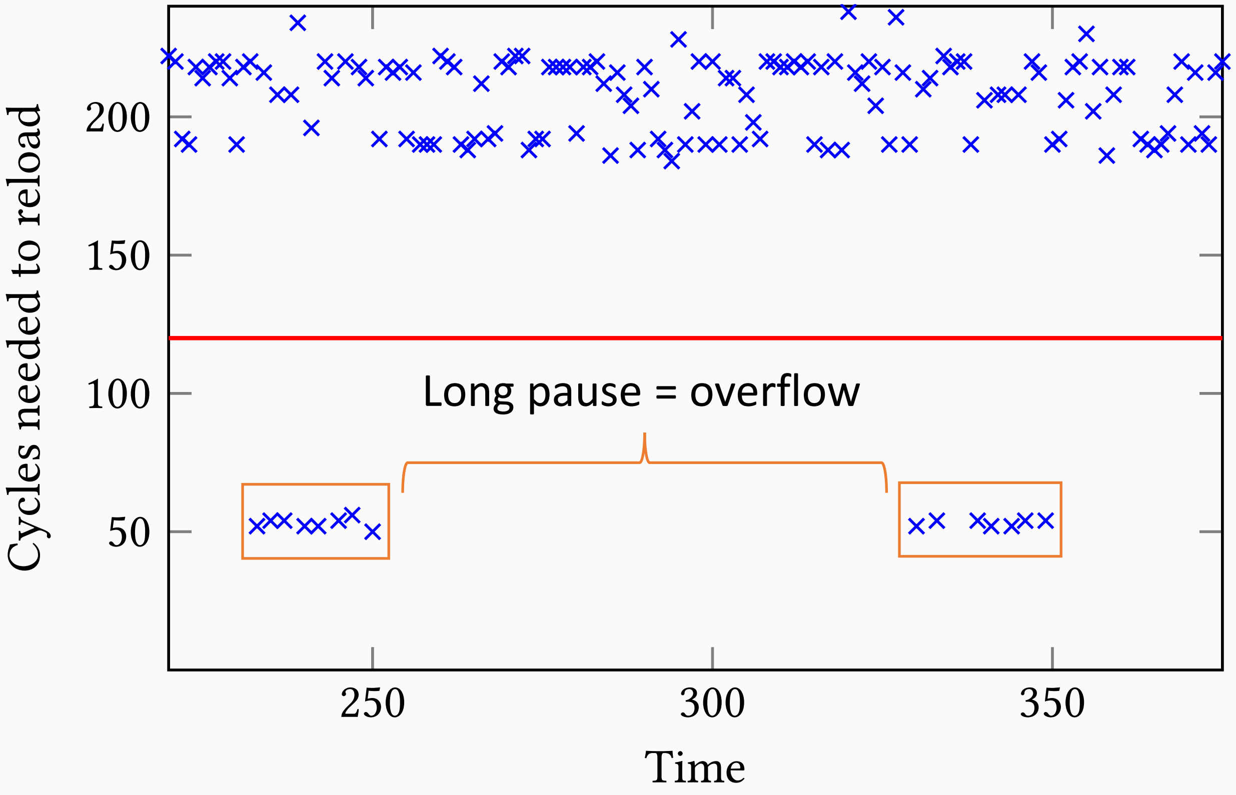


Long pause = overflow

Trace Interpretation



Trace Interpretation



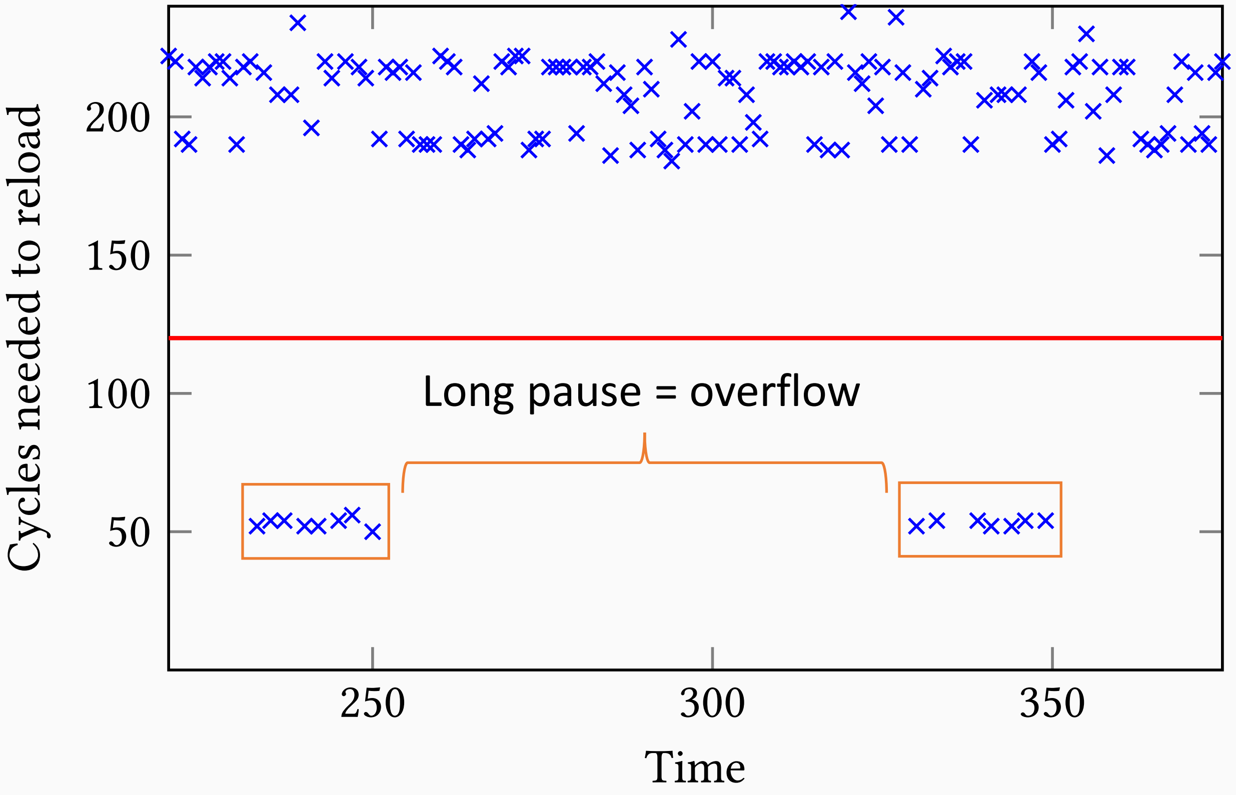
Rules ($b \in \{0,1\}$):

Vvvv -> $111b$

Vvvvv -> $yyyyb$, $yyyy \in \{110b, 10bb, 0111\}$

Vv.....v -> $0 \dots 0yyyyb$

Trace Interpretation



Rules ($b \in \{0,1\}$):

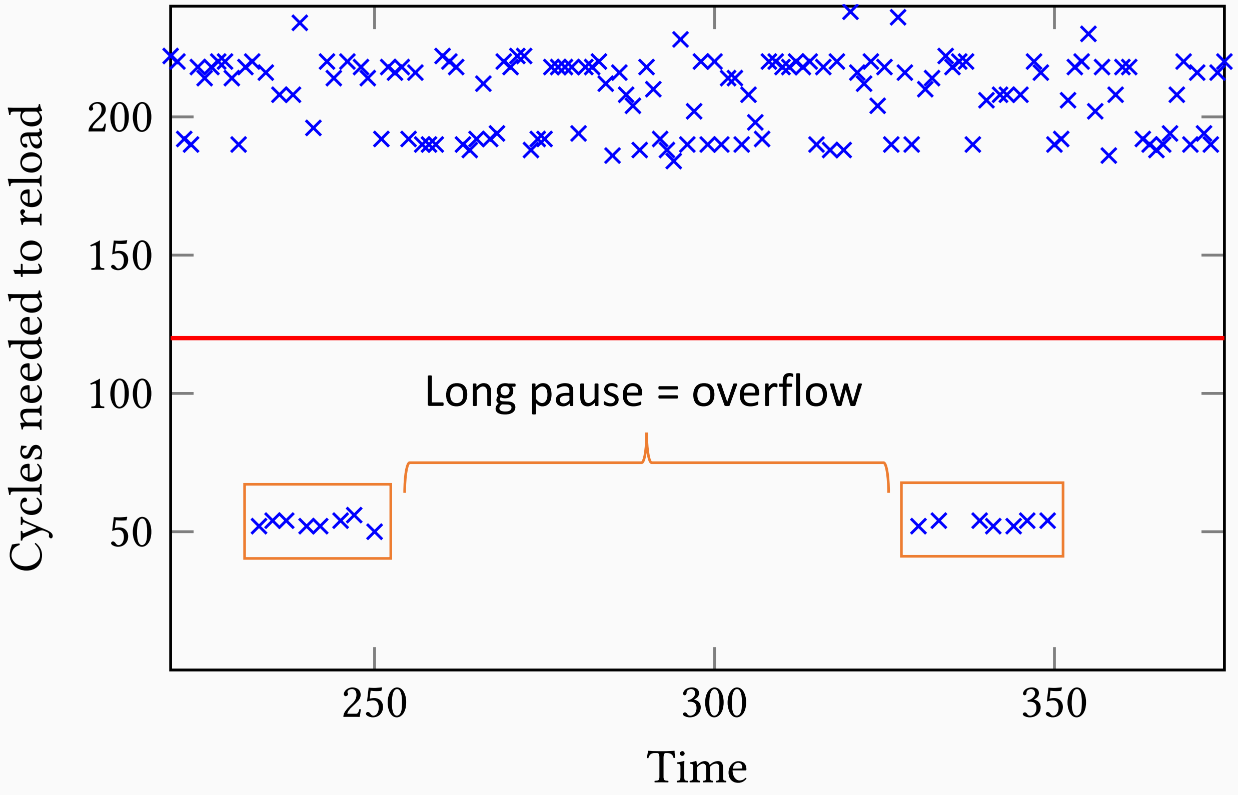
Vvvv -> $111b$

Vvvvv -> $yyyyb, yyyy \in \{110b, 10bb, 0111\}$

Vv.....v -> $0 \ldots 0yyyyb$

Vvvv Vvvvv Vvvvvv Vvvvv Vvvvv Vvvvv Vvvv

Trace Interpretation



Rules ($b \in \{0,1\}$):

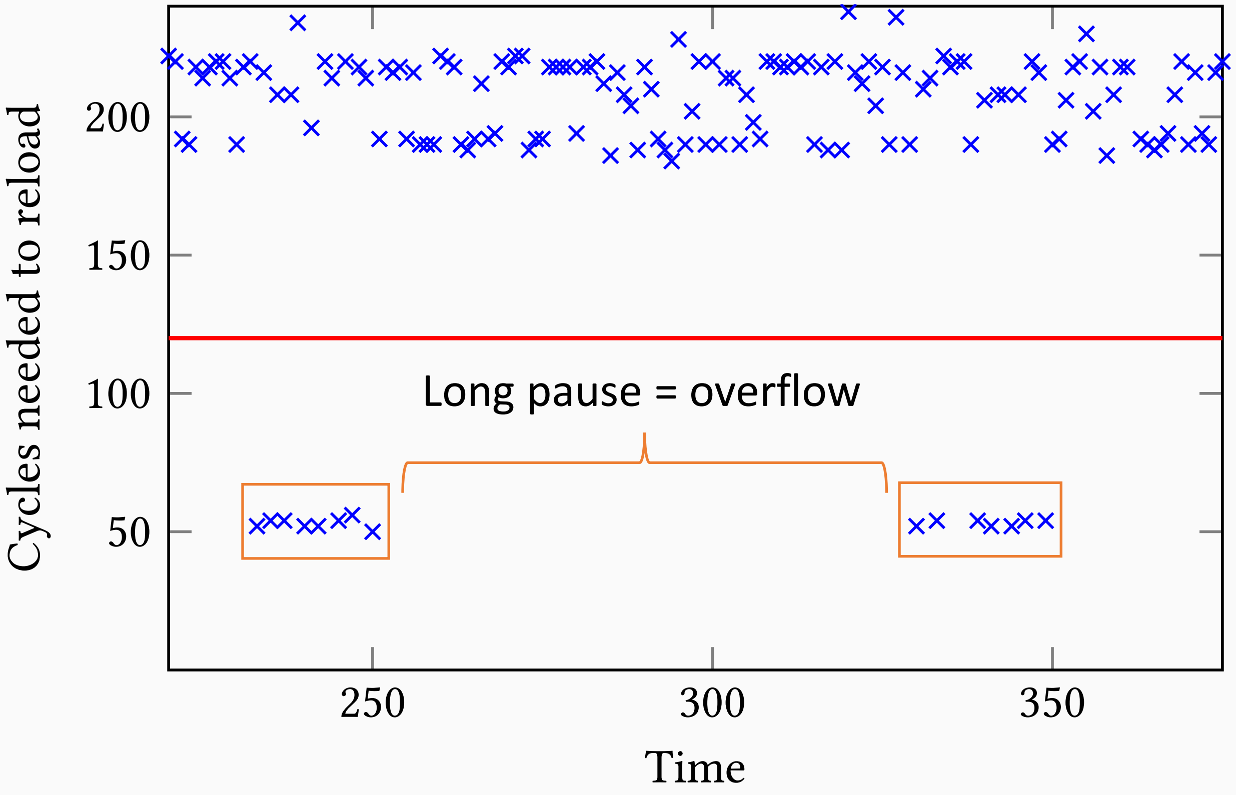
Vvvv -> $111b$

Vvvvv -> $yyyyb, yyyy \in \{110b, 10bb, 0111\}$

Vv.....v -> $0 \ldots 0yyyyb$

Vvvv Vvvvv Vvvvvv Vvvvv Vvvvv Vvvvv Vvvv
111b

Trace Interpretation



Rules ($b \in \{0,1\}$):

Vvvv -> $111b$

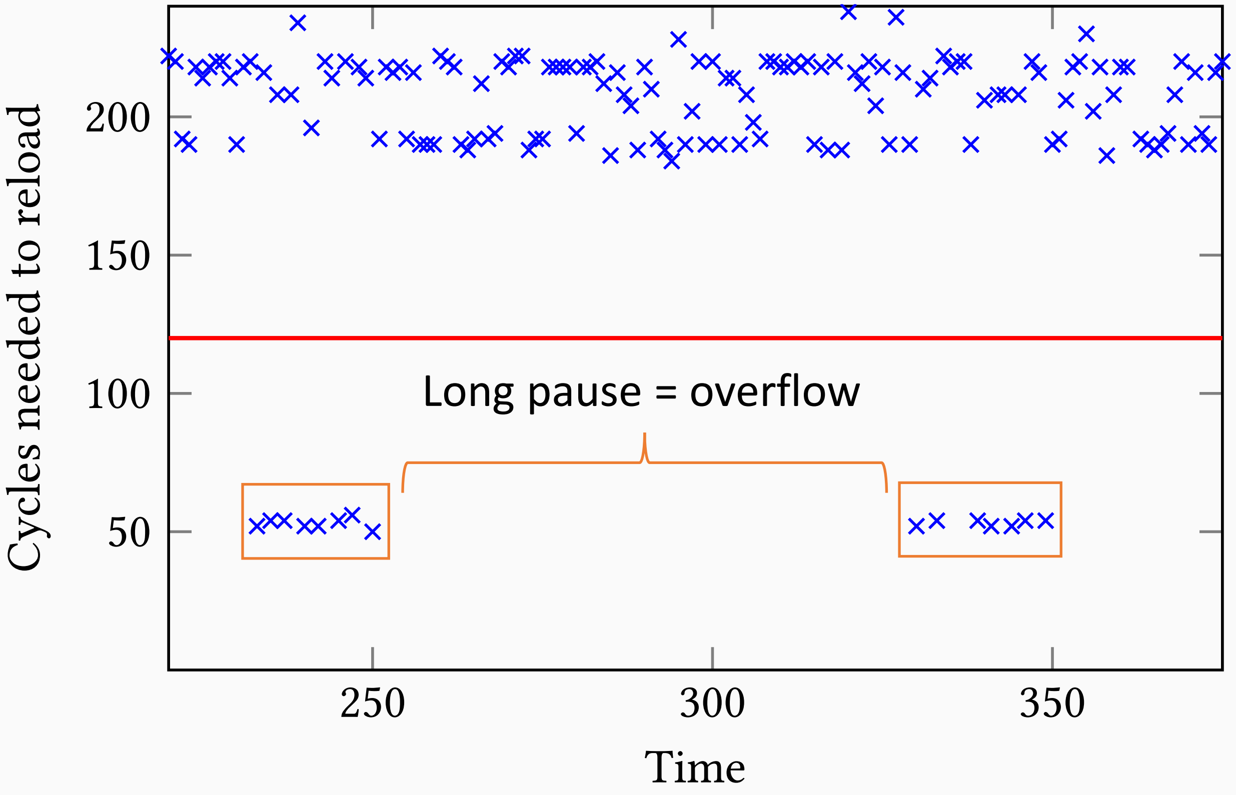
Vvvvv -> $yyyyb$, $yyyy \in \{110b, 10bb, 0111\}$

Vv.....v -> $0 \ldots 0yyyyb$

Vvvv Vvvvv Vvvvvv Vvvvv Vvvvv Vvvvv Vvvv

111b yyyyb

Trace Interpretation



Rules ($b \in \{0,1\}$):

Vvvv -> $111b$

Vvvvv -> $yyyyb$, $yyyy \in \{110b, 10bb, 0111\}$

Vv.....v -> $0 \ldots 0yyyyb$

Vvvv Vvvvv Vvvvvv Vvvvv Vvvvv Vvvvv Vvvv
111b yyyyb 0yyyyb yyyyb yyyyb yyyyb bbbb

Dictionary Attack

Client : $x = H(salt \mid\mid H(user\_id:password))$
$v = g^x \bmod p$

Dictionary Attack

Client : x = H(salt || H(user_id:password))
v = g^x mod p

$b \in \{0,1\}$
$yyyy \in \{110b, 10bb, 0111\}$

Recovered: `1 1 1 b y y y y b 0 y y y y b 1 1 1 b 0 y y y y b`

Dictionary Attack

Client : x = H(salt || H(user_id:password))
v = g^x mod p

$b \in \{0,1\}$
$yyyy \in \{110b, 10bb, 0111\}$

```
Recovered:      1 1 1 b y y y y b 0 y y y y b 1 1 1 b 0 y y y y b

   pwd_1        1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 1
   pwd_2        1 1 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1
   pwd_3        0 1 1 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0
   pwd_4        1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1
   pwd_5        0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0
   …
   pwd_n        1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1
```

Password	**X value**

Dictionary Attack

Client : x = H(salt || H(user_id:password))
v = g^x mod p

$b \in \{0,1\}$
$yyyy \in \{110b, 10bb, 0111\}$

```
Recovered:      1 1 1 b y y y y b 0 y y y y b 1 1 1 b 0 y y y y b

   pwd_1        1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 1
   pwd_2        1 1 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1
   pwd_3        0 1 1 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0
   pwd_4        1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1
   pwd_5        0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0
   …
   pwd_n        1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1
```

Password	X value

Dictionary Attack

Client : x = H(salt || H(user_id:password))
v = g^x mod p

$b \in \{0,1\}$
$yyyy \in \{110b, 10bb, 0111\}$

Recovered:	1	1	1	b	y	y	y	y	b	0	y	y	y	y	b	1	1	1	b	0	y	y	y	y	b
pwd_1	1	0	1	0	0	0	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1
pwd_2	1	1	0	0	1	0	1	1	1	1	1	1	1	0	0	0	0	0	1	0	1	1	1	0	1
pwd_3	0	1	1	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	1	0	0	0
pwd_4	1	1	1	1	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0	0	0	1	1	1	1
pwd_5	0	1	1	1	1	0	1	1	1	1	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0
…																									
pwd_n	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	0	1	0	1

Password	**X value**

Dictionary Attack

Client : x = H(salt || H(user_id:password))
v = g^x mod p

$b \in \{0,1\}$
$yyyy \in \{110b, 10bb, 0111\}$

Password	X value	Diff score
Recovered:	1 1 1 b y y y y b 0 y y y y b 1 1 1 b 0 y y y y b	
pwd_1	1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 1 1	15
pwd_2	1 1 0 0 1 0 1 1 1 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 1	14
pwd_3	0 1 1 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 1 0 0 0	11
pwd_4	1 1 1 1 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1	**0**
pwd_5	0 1 1 1 1 0 1 1 1 1 0 0 1 0 1 1 1 0 0 0 0 1 0 0 0	11
…		
pwd_n	1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 1	12

Single Measurement Attack

- Very accurate measurement
- Each bit of information halves the number of possible passwords
    - k bits of information => 2^{-k} probability of false positive/negative

Single Measurement Attack

- Very accurate measurement
- Each bit of information halves the number of possible passwords
  - k bits of information => 2^{-k} probability of false positive/negative

For a n bits exponent, we get $\mathrm{k} = 0{,}4n + 2$ bits on average (verified empirically)

Single Measurement Attack

- Very accurate measurement
- Each bit of information halves the number of possible passwords
    - k bits of information => 2^{-k} probability of false positive/negative

For a n bits exponent, we get $\mathrm{k} = 0{,}4n + 2$ bits on average (verified empirically)

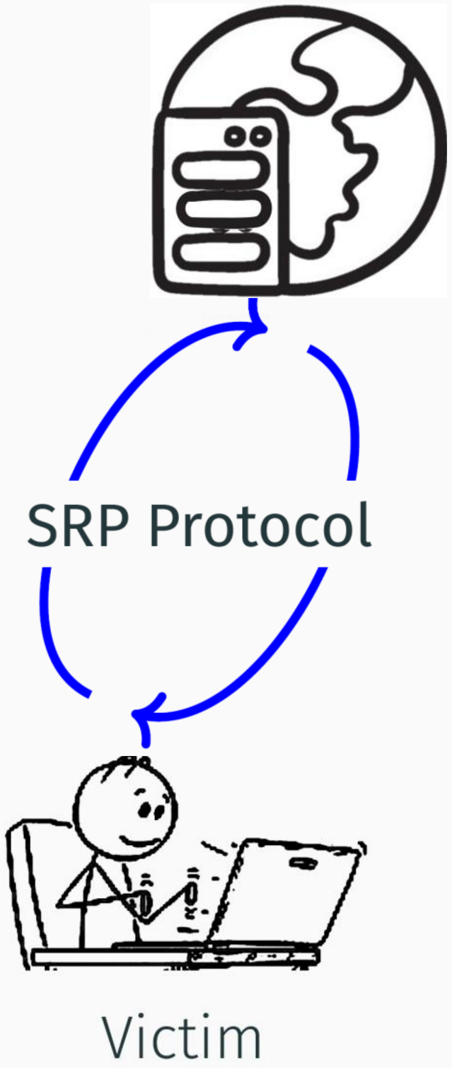
SHA-1: 66 bits of information

SHA-256: 104 bits of information

Attacker Model

- Unprivileged spyware on the victim station
- Victim tries to connect
- MitM can help to gather more information (optional)

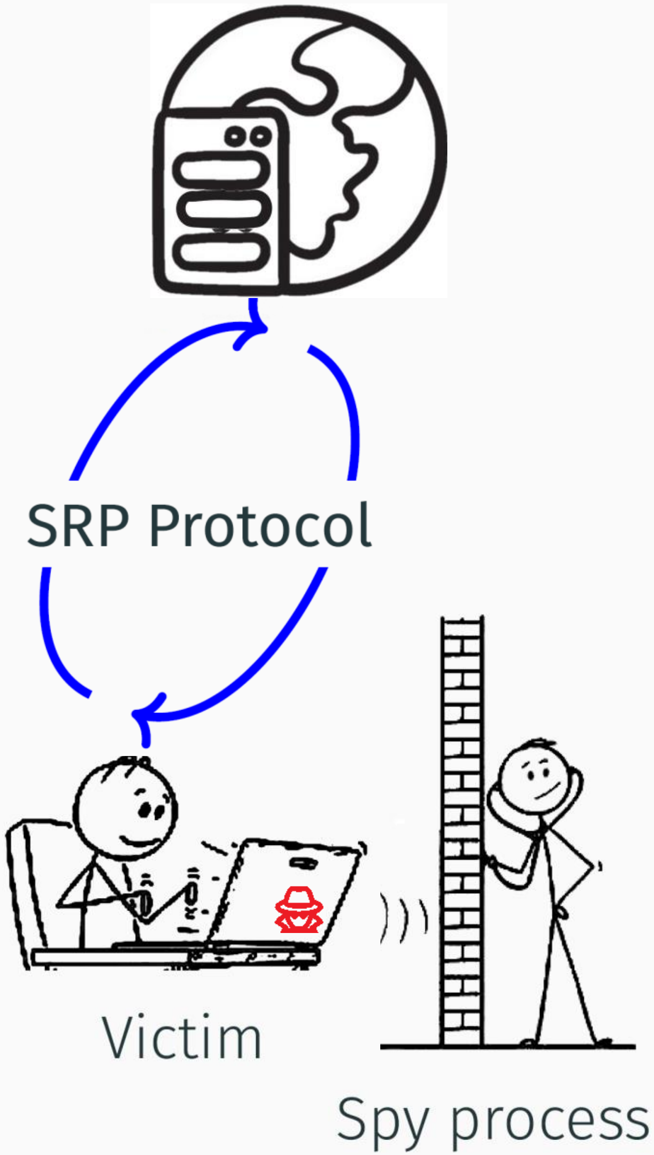
Classical Workflow



SRP Protocol

Victim

Classical Workflow

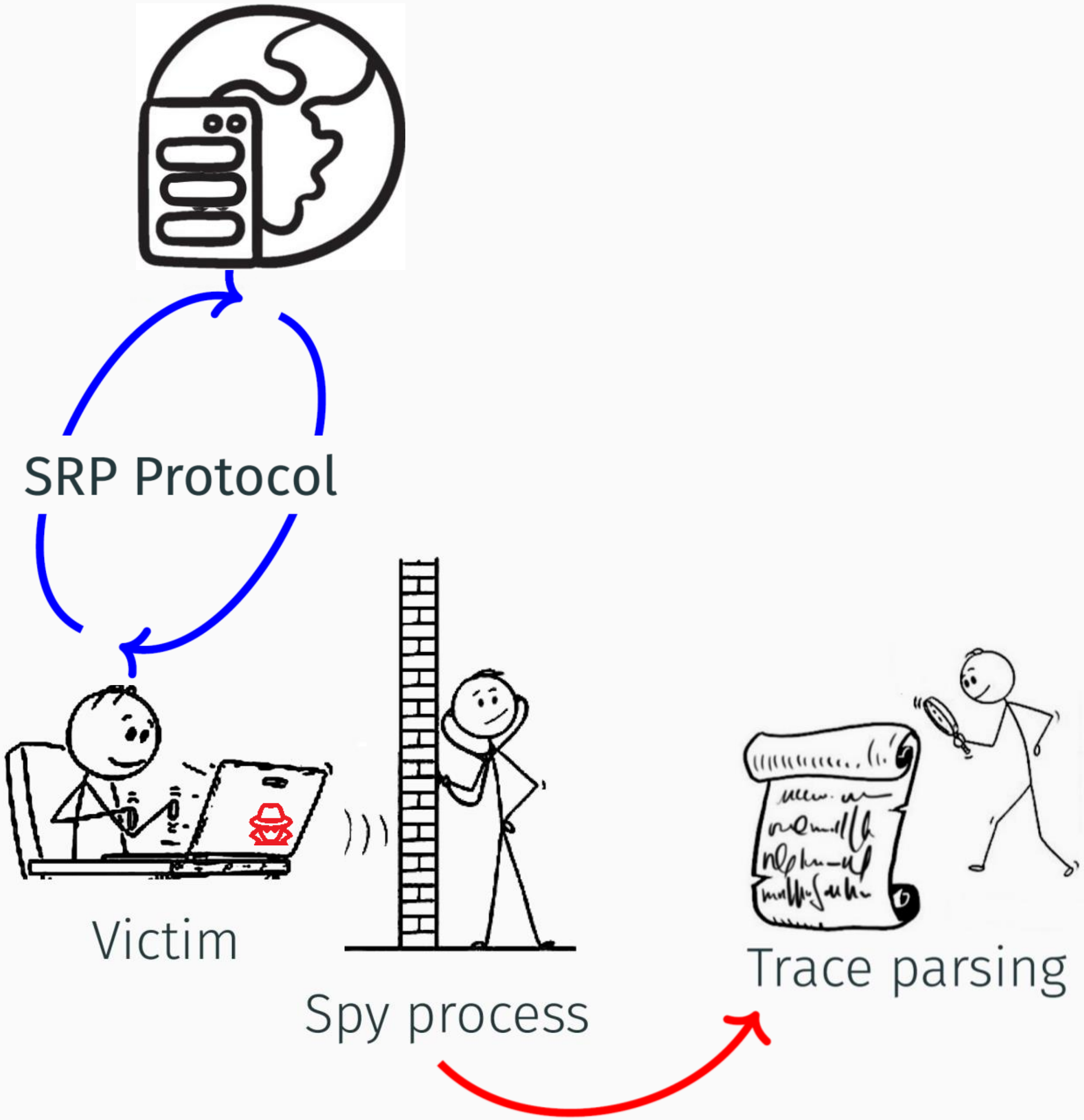


SRP Protocol

Victim

Spy process

Classical Workflow



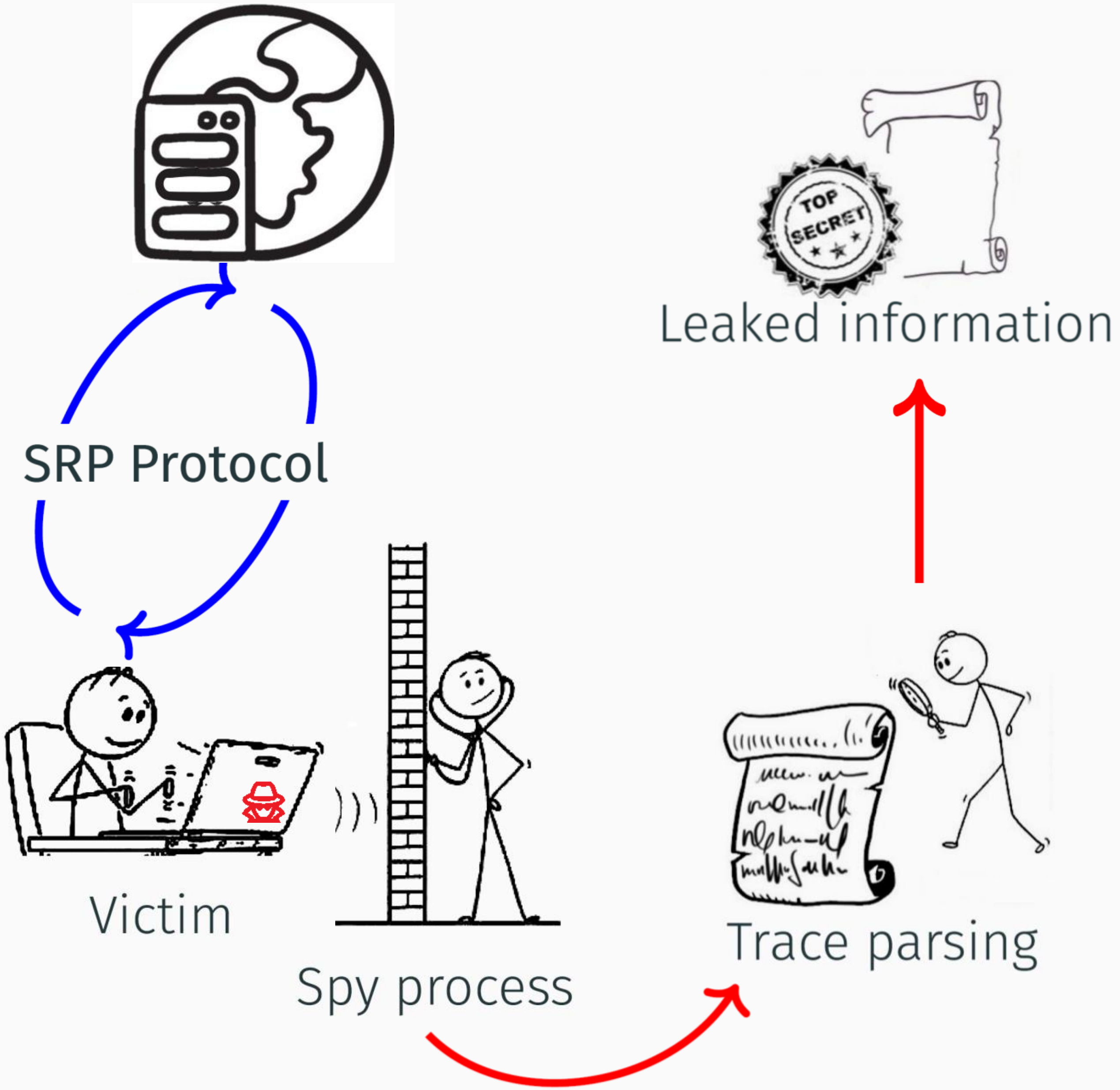
SRP Protocol

Victim

Spy process

Trace parsing

Classical Workflow



Leaked information

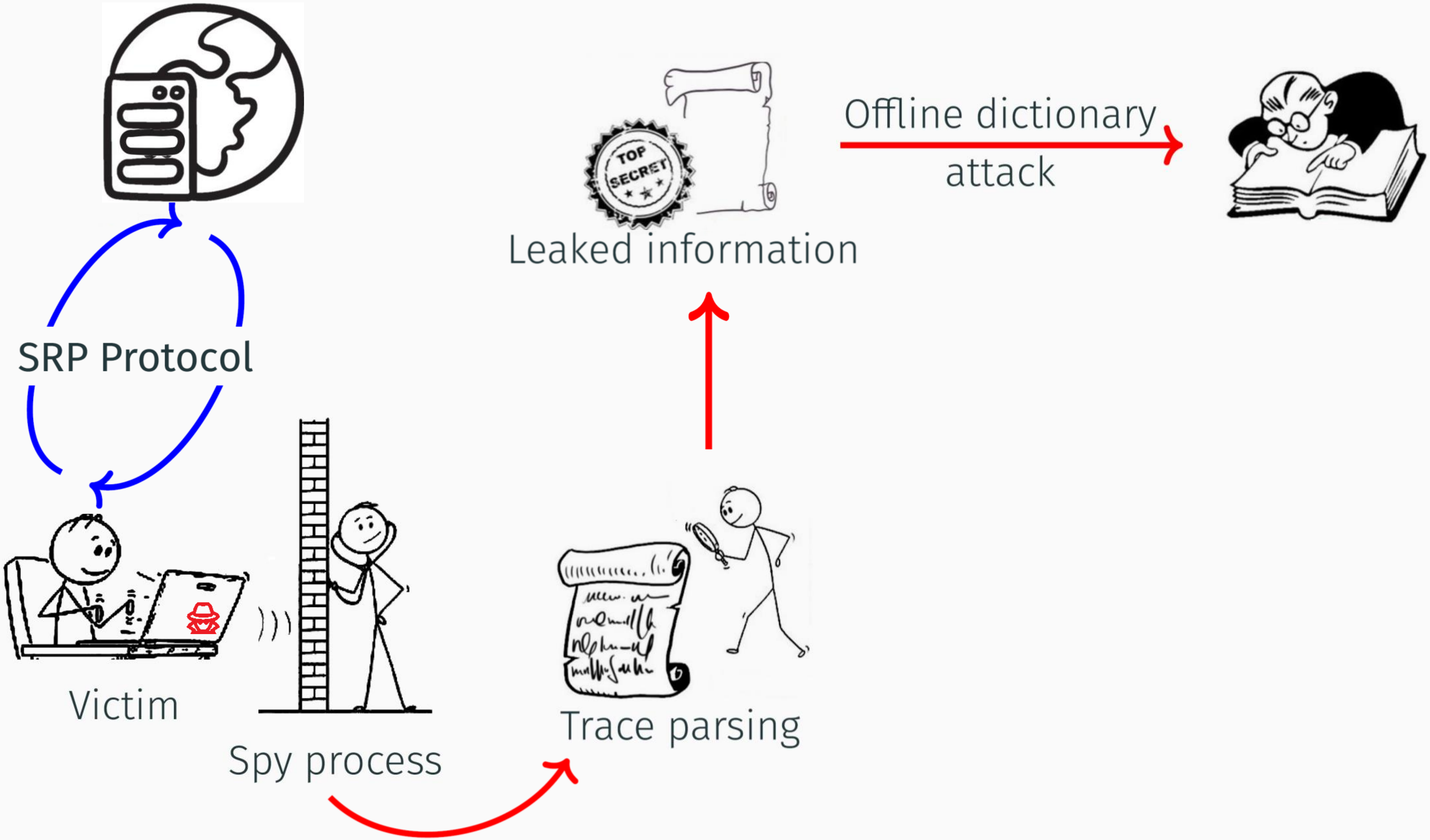
SRP Protocol

Victim

Spy process

Trace parsing

Classical Workflow



SRP Protocol

Victim

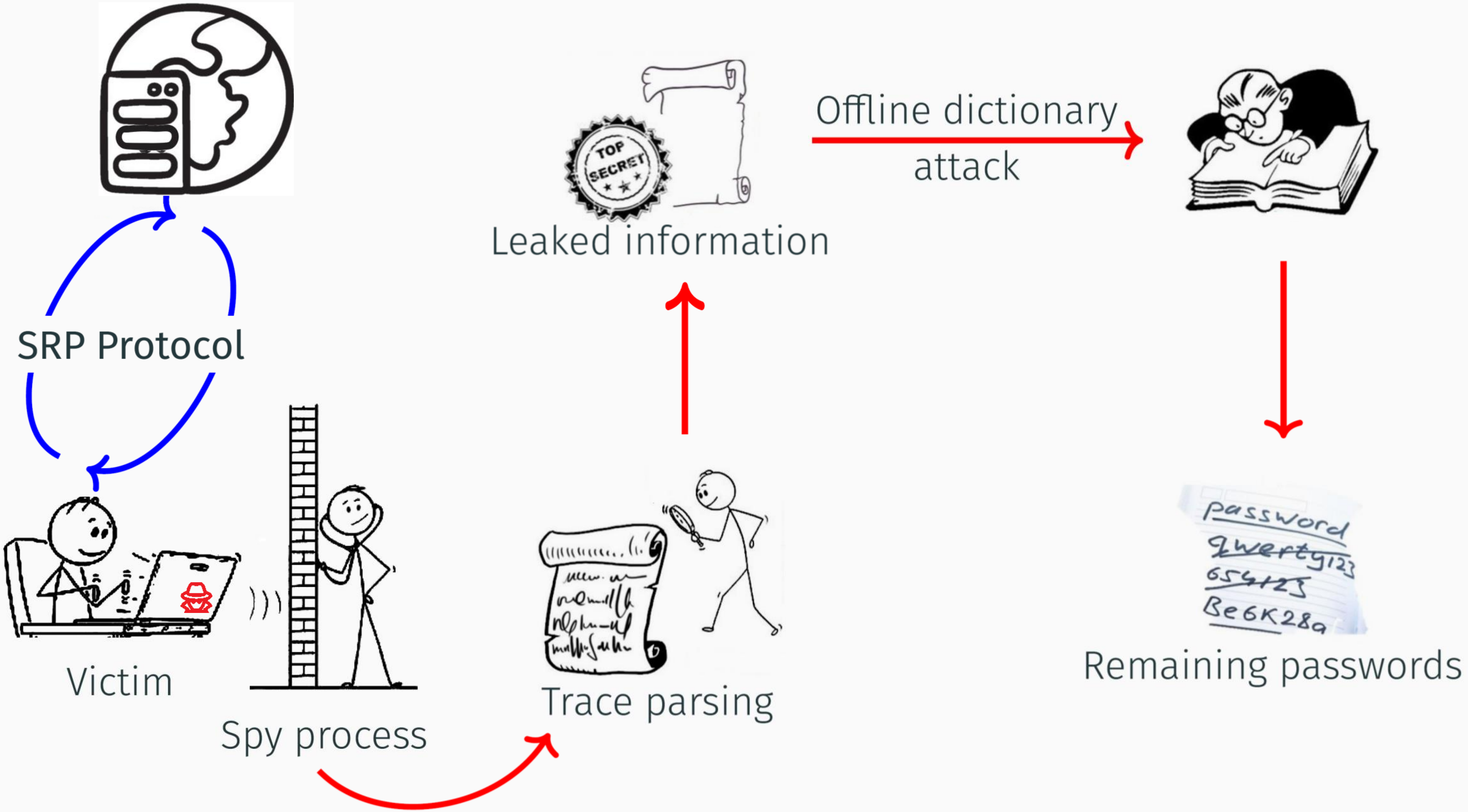
Spy process

Trace parsing

Leaked information

Offline dictionary attack

Classical Workflow



SRP Protocol

Victim

Spy process

Trace parsing

Leaked information

Offline dictionary attack

Remaining passwords

Practical Impact

Impacted Projects

- Lots of project using OpenSSL are impacted, including
  - OpenSSL TLS-SRP
  - Apple HomeKit ADK
  - PySRP (used in ProtonMail python client)
  - …

Impacted Projects

- Lots of project using OpenSSL are impacted, including
  - OpenSSL TLS-SRP
  - Apple HomeKit ADK
  - PySRP (used in ProtonMail python client)
  - …

Wait, how are big numbers managed in high level languages ?...

Impacted Languages

- Many reference libraries are based on OpenSSL to manage bignums
- They usually (never ?) manage the flag properly
  - Ruby/openssl
  - Javascript node-bignum
  - Erlang OTP

All SRP implementations using these packages / libraries would be affected!

Mitigations & Conclusion

Mitigations

Two choices:

- Patch this particular issue by adding the proper flag
  - Most projects use the bignum API, not the whole SRP
  - Difficult to propagate
  - Root cause remains

- Switch to a secure by default implementation (flag for insecure/optimized)
  - No flag = secure implementation (potential performance loss)
  - All projects are patched at once

Mitigations

Two choices:

- Patch this particular issue by adding the proper flag ← OpenSSL's choice
    - Most projects use the bignum API, not the whole SRP
    - **Difficult to propagate**
    - Root cause remains

- Switch to a secure by default implementation (flag for insecure/optimized)
    - No flag = secure implementation (potential performance loss)
    - All projects are patched at once

Conclusion

- Practical attack against SRP implementations
  - Vulnerability inherited by lots of projects
  - Easy to exploit because we can use each recover bits independently

Long term lesson: be careful with SCA, especially in PAKE implementation

Conclusion

- Practical attack against SRP implementations
  - Vulnerability inherited by lots of projects
  - Easy to exploit because we can use each recover bits independently

Long term lesson: be careful with SCA, especially in PAKE implementation

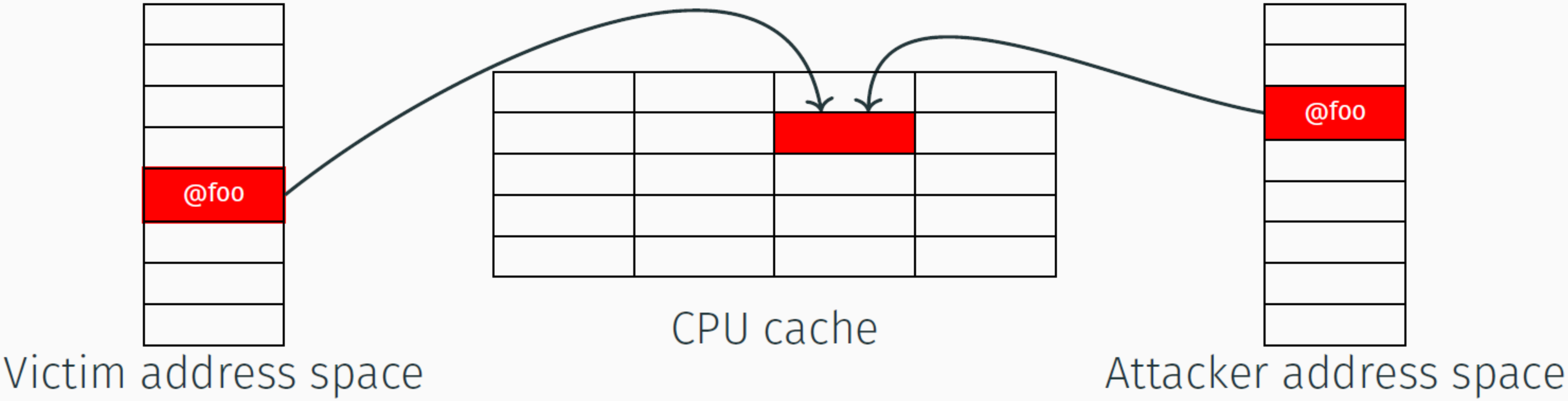
- Leakage in a weak generic function
  - Other protocols with small base may also use it
  - Contact use if you think of one!

Thank you for your attention!

https://gitlab.inria.fr/ddealmei/poc-openssl-srp

@ daniel.de-almeida-braga@irisa.fr

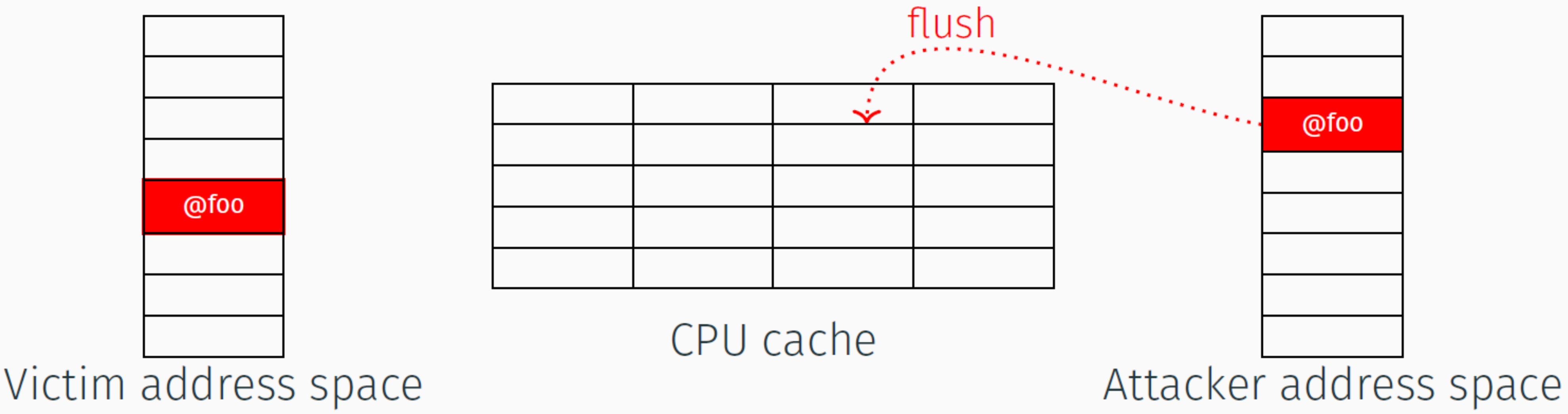
FLUSH+RELOAD[1]



1. Maps the victim's address space

[1] Y. Yarom et al. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium. 2014

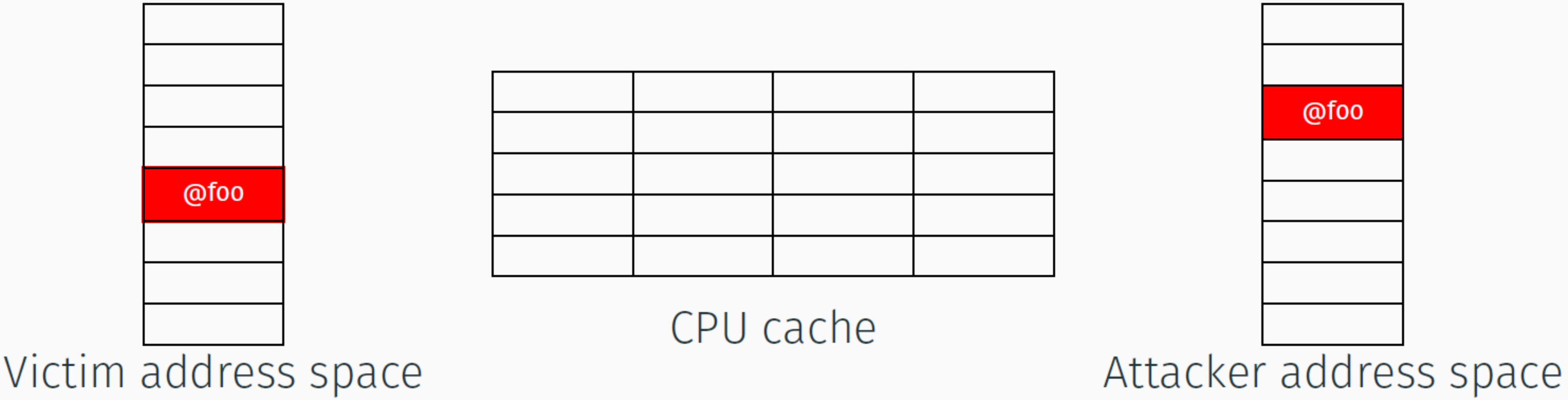
FLUSH+RELOAD[1]



1. Maps the victim's address space
2. Flush the instruction we monitor

[1] Y. Yarom et al. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium. 2014

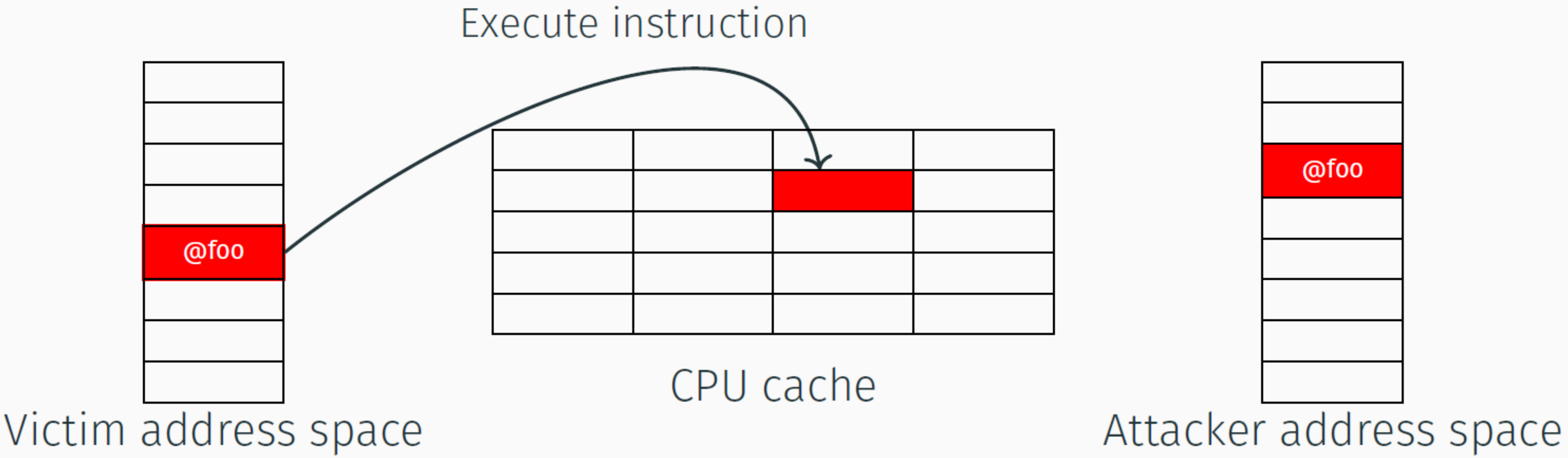
FLUSH+RELOAD[1]



1. Maps the victim’s address space
2. Flush the instruction we monitor
3. See how much time it takes to reload

[1] Y. Yarom et al. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium. 2014

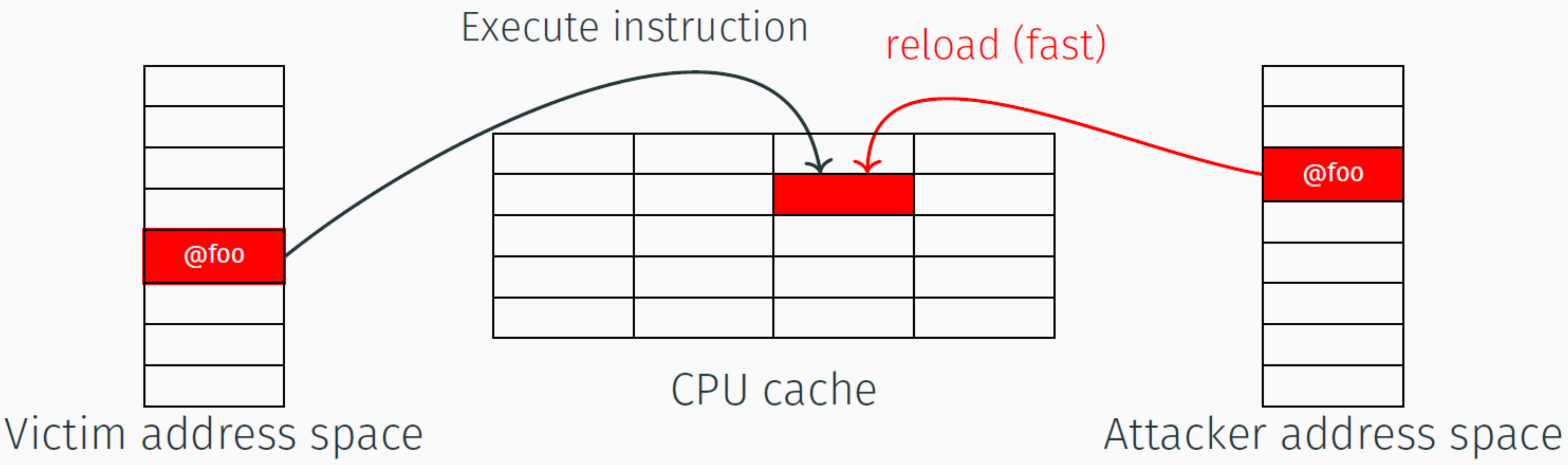
FLUSH+RELOAD[1]



1. Maps the victim's address space
2. Flush the instruction we monitor
3. See how much time it takes to reload

[1] Y. Yarom et al. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium. 2014

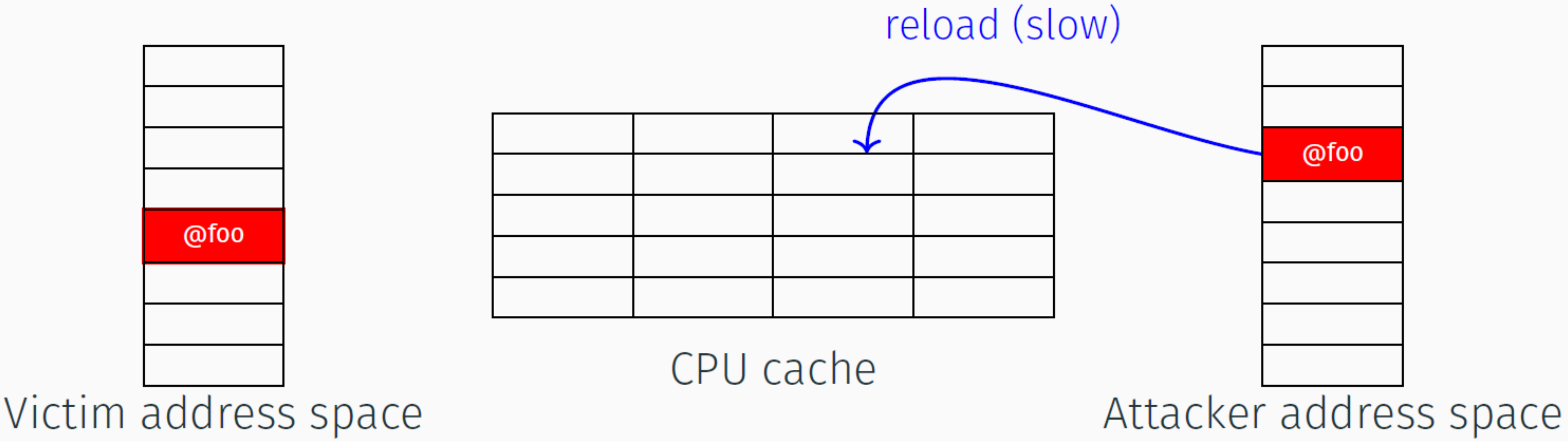
FLUSH+RELOAD[1]



1. Maps the victim's address space
2. Flush the instruction we monitor
3. See how much time it takes to reload
    - Fast ⇒ the victim already executed

[1] Y. Yarom et al. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium. 2014

FLUSH+RELOAD[1]



1. Maps the victim's address space
2. Flush the instruction we monitor
3. See how much time it takes to reload
   - Fast ⇒ the victim already executed
   - Slow ⇒ the victim did not

[1] Y. Yarom et al. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium. 2014