Constant-time square-and-multiply

D. J. Bernstein

University of Illinois at Chicago;
Ruhr University Bochum

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    if 1&(e>>i):
      y = y*x
  return y
```


This code uses 256 squarings, plus 1 extra multiplication for each bit set in e.

Problem when e is secret: time leaks number of bits set in e.

Constant-time square-and-multiply

D. J. Bernstein

University of Illinois at Chicago;
Ruhr University Bochum

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    if 1&(e>>i):
      y = y*x
  return y
```


This code uses 256 squarings, plus 1 extra multiplication for each bit set in e.

Problem when e is secret: time leaks number of bits set in e.

"I'll choose secret 256-bit e with exactly 128 bits set. There are enough of these e, and then there are no more leaks."

Constant-time square-and-multiply

D. J. Bernstein

University of Illinois at Chicago;
Ruhr University Bochum

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    if 1&(e>>i):
      y = y*x
  return y
```


This code uses 256 squarings, plus 1 extra multiplication for each bit set in e.

Problem when e is secret: time leaks number of bits set in e.

"I'll choose secret 256-bit e with exactly 128 bits set. There are enough of these e, and then there are no more leaks."

— Time still depends on e, even if each multiplication takes time independent of inputs.

t-time
nd-multiply

ernstein

ty of Illinois at Chicago;
niversity Bochum

```
256bit(x,e):

 in reversed(range(256)):
 y*y
1&(e>>i):
 = y*x
n y
```


This code uses 256 squarings,
plus 1 extra multiplication
for each bit set in e.

Problem when e is secret: time
leaks number of bits set in e.

"I'll choose secret 256-bit e with
exactly 128 bits set. There are
enough of these e, and then
there are no more leaks."

— Time still depends on e,
even if each multiplication
takes time independent of inputs.

Hardwar
is inhere

CPU des

ly

is at Chicago;
ochum

```
e):

sed(range(256)):
```


This code uses 256 squarings,
plus 1 extra multiplication
for each bit set in e.

Problem when e is secret: time
leaks number of bits set in e.

"I'll choose secret 256-bit e with
exactly 128 bits set. There are
enough of these e, and then
there are no more leaks."

— Time still depends on e,
even if each multiplication
takes time independent of inputs.

Hardware reality:
is inherently exper

CPU designers try

ago;

---

e(256)):

This code uses 256 squarings,
plus 1 extra multiplication
for each bit set in e.

Problem when e is secret: time
leaks number of bits set in e.

"I'll choose secret 256-bit e with
exactly 128 bits set. There are
enough of these e, and then
there are no more leaks."

— Time still depends on e,
even if each multiplication
takes time independent of inputs.

Hardware reality: Accessing
is inherently expensive.

CPU designers try to reduce

This code uses 256 squarings,
plus 1 extra multiplication
for each bit set in e.

Problem when e is secret: time
leaks number of bits set in e.

"I'll choose secret 256-bit e with
exactly 128 bits set. There are
enough of these e, and then
there are no more leaks."

— Time still depends on e,
even if each multiplication
takes time independent of inputs.

Hardware reality: Accessing RAM
is inherently expensive.

CPU designers try to reduce cost.

This code uses 256 squarings, plus 1 extra multiplication for each bit set in e.

Problem when e is secret: time leaks number of bits set in e.

"I'll choose secret 256-bit e with exactly 128 bits set. There are enough of these e, and then there are no more leaks."

— Time still depends on e, even if each multiplication takes time independent of inputs.

Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost.

Example: "L1 cache" typically has 32KB of recently used data.

This cache inspects RAM addresses, performs various computations on addresses to try to save time.

This code uses 256 squarings, plus 1 extra multiplication for each bit set in e.

Problem when e is secret: time leaks number of bits set in e.

"I'll choose secret 256-bit e with exactly 128 bits set. There are enough of these e, and then there are no more leaks."

— Time still depends on e, even if each multiplication takes time independent of inputs.

Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost.

Example: "L1 cache" typically has 32KB of recently used data.

This cache inspects RAM addresses, performs various computations on addresses to try to save time.

. . . so time is a function of RAM addresses. Avoid all data flow from secrets to RAM addresses.

de uses 256 squarings,
xtra multiplication
bit set in e.

when e is secret: time
mber of bits set in e.

ose secret 256-bit e with
128 bits set. There are
of these e, and then
e no more leaks."

still depends on e,
each multiplication
me independent of inputs.

Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost.

Example: "L1 cache" typically has 32KB of recently used data.

This cache inspects RAM addresses, performs various computations on addresses to try to save time.

. . . so time is a function of RAM addresses. Avoid all data flow from secrets to RAM addresses.

Example
from sec

Often de
for softw
the same

6 squarings,
plication
e.

s secret: time
its set in e.

256-bit e with
et. There are
, and then
leaks."

nds on e,
plication
ndent of inputs.

Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost.

Example: "L1 cache" typically has 32KB of recently used data.

This cache inspects RAM addresses, performs various computations on addresses to try to save time.

. . . so time is a function of RAM addresses. Avoid all data flow from secrets to RAM addresses.

Example: Avoid a
from secrets to bra

Often described as
for software, but c
the same hardware

s,

ime
e.

with
are

nputs.

Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost.

Example: "L1 cache" typically has 32KB of recently used data.

This cache inspects RAM addresses, performs various computations on addresses to try to save time.

. . . so time is a function of RAM addresses. Avoid all data flow from secrets to RAM addresses.

Example: Avoid all data flow from secrets to branch cond

Often described as a separat for software, but comes from the same hardware reality.

Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost.

Example: “L1 cache” typically has 32KB of recently used data.

This cache inspects RAM addresses, performs various computations on addresses to try to save time.

. . . so time is a function of RAM addresses. Avoid all data flow from secrets to RAM addresses.

Example: Avoid all data flow from secrets to branch conditions.

Often described as a separate rule for software, but comes from the same hardware reality.

Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost.

Example: "L1 cache" typically has 32KB of recently used data.

This cache inspects RAM addresses, performs various computations on addresses to try to save time.

. . . so time is a function of RAM addresses. Avoid all data flow from secrets to RAM addresses.

Example: Avoid all data flow from secrets to branch conditions.

Often described as a separate rule for software, but comes from the same hardware reality.

How CPU runs a program (example of "code = data"):

```
while True:
  insn = RAM[state.ip]
  state = execute(state,insn)
```

`ip` ("instruction pointer" or "program counter"): address in RAM of next instruction.

3

e reality: Accessing RAM
ntly expensive.

signers try to reduce cost.

e: "L1 cache" typically
B of recently used data.

che inspects RAM
es, performs various
ations on addresses
o save time.

me is a function of RAM
es. Avoid all data flow
crets to RAM addresses.

4

Example: Avoid all data flow
from secrets to branch conditions.

Often described as a separate rule
for software, but comes from
the same hardware reality.

How CPU runs a program
(example of "code = data"):

```
while True:
  insn = RAM[state.ip]
  state = execute(state,insn)
```

`ip` ("instruction pointer" or
"program counter"): address
in RAM of next instruction.

Standard
to follow
Square a

```
def pow
  y = 1
  for i
    y =
    yx =
    bit
    y =
  retur
```

If `bit` is
an unuse

Accessing RAM
nsive.

to reduce cost.

he" typically
tly used data.

ts RAM
ns various
addresses
e.

nction of RAM
ll data flow
AM addresses.

Example: Avoid all data flow from secrets to branch conditions.

Often described as a separate rule for software, but comes from the same hardware reality.

How CPU runs a program (example of "code = data"):

```
while True:
  insn = RAM[state.ip]
  state = execute(state,insn)
```

`ip` ("instruction pointer" or "program counter"): address in RAM of next instruction.

Standard square-a
to follow these dat
Square and always

```
def pow256bit(x,
  y = 1
  for i in rever
    y = y*y
    yx = y*x
    bit = 1&(e>>
    y = y+(yx-y)
  return y
```

If `bit` is 0 then `yx`
an unused "dumm

RAM

cost.

lly
lata.

RAM
ow
sses.

Example: Avoid all data flow from secrets to branch conditions.

Often described as a separate rule for software, but comes from the same hardware reality.

How CPU runs a program (example of "code = data"):

```
while True:
  insn = RAM[state.ip]
  state = execute(state,insn)
```

`ip` ("instruction pointer" or "program counter"): address in RAM of next instruction.

Standard square-and-multipl
to follow these data-flow rul
Square and always multiply.

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range
    y = y*y
    yx = y*x
    bit = 1&(e>>i)
    y = y+(yx-y)*bit
  return y
```

If `bit` is 0 then `yx` computa
an unused "dummy operatio

Example: Avoid all data flow from secrets to branch conditions.

Often described as a separate rule for software, but comes from the same hardware reality.

How CPU runs a program (example of “code = data”):

```
while True:
  insn = RAM[state.ip]
  state = execute(state,insn)
```

`ip` (“instruction pointer” or “program counter”): address in RAM of next instruction.

Standard square-and-multiply fix to follow these data-flow rules: Square and always multiply.

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    yx = y*x
    bit = 1&(e>>i)
    y = y+(yx-y)*bit
  return y
```

If `bit` is 0 then `yx` computation is an unused “dummy operation”.

e: Avoid all data flow
crets to branch conditions.

escribed as a separate rule
vare, but comes from
e hardware reality.

U runs a program
e of “code = data”):

```
rue:
= RAM[state.ip]
 = execute(state,insn)
```

truction pointer” or
n counter”): address
 of next instruction.

Standard square-and-multiply fix
to follow these data-flow rules:
Square and always multiply.

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    yx = y*x
    bit = 1&(e>>i)
    y = y+(yx-y)*bit
  return y
```

If `bit` is 0 then `yx` computation is
an unused “dummy operation”.

Another

```
def pow
  y,i,j
  while
    if
      y
      i
    
      e
    
    els
      y
      i
  retur
```

ll data flow
anch conditions.

s a separate rule
comes from
e reality.

program
e = data”):

```
te.ip]
e(state,insn)
```

ointer” or
”): address
struction.

Standard square-and-multiply fix to follow these data-flow rules: Square and always multiply.

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    yx = y*x
    bit = 1&(e>>i)
    y = y+(yx-y)*bit
  return y
```

If `bit` is 0 then `yx` computation is an unused “dummy operation”.

Another approach,

```
def pow256bit(x,
  y,i,j = 1,255,
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,
  return y
```

w
itions.
te rule
n
):
nsn)
s

Standard square-and-multiply fix to follow these data-flow rules: Square and always multiply.

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    yx = y*x
    bit = 1&(e>>i)
    y = y+(yx-y)*bit
  return y
```

If `bit` is 0 then `yx` computation is an unused “dummy operation”.

Another approach, not well

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i):
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,0
  return y
```

Standard square-and-multiply fix to follow these data-flow rules: Square and always multiply.

```
def pow256bit(x,e):
  y = 1
  for i in reversed(range(256)):
    y = y*y
    yx = y*x
    bit = 1&(e>>i)
    y = y+(yx-y)*bit
  return y
```

If `bit` is 0 then `yx` computation is an unused “dummy operation”.

Another approach, not well known:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i):
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,0
  return y
```

d square-and-multiply fix
/ these data-flow rules:
and always multiply.

```
256bit(x,e):

 in reversed(range(256)):
 y*y
= y*x
 = 1&(e>>i)
 y+(yx-y)*bit
n y
```

0 then yx computation is
ed “dummy operation”.

Another approach, not well known:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i):
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,0
  return y
```

This is l
original s

j is “inst
0 if at to
1 if in m

Each “in
includes

nd-multiply fix
ta-flow rules:
; multiply.

```
e):

sed(range(256)):

i)
*bit
```

x computation is
y operation".

Another approach, not well known:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i):
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,0
  return y
```

This is like CPU's
original square-and

j is "instruction po
0 if at top of loop
1 if in middle of lo

Each "instruction"
includes exactly on

y fix
es:

(256)):

tion is
on".

Another approach, not well known:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i):
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,0
  return y
```

This is like CPU's perspectiv
original square-and-multiply.

j is "instruction pointer":
0 if at top of loop,
1 if in middle of loop.

Each "instruction" here
includes exactly one multiply

Another approach, not well known:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i):
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,0
  return y
```


This is like CPU's perspective on original square-and-multiply.

j is "instruction pointer":
0 if at top of loop,
1 if in middle of loop.

Each "instruction" here includes exactly one multiply.

Another approach, not well known:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    if j == 0:
      y = y*y
      if 1&(e>>i):
        j = 1
      else:
        i = i-1
    else:
      y = y*x
      i,j = i-1,0
  return y
```


This is like CPU's perspective on original square-and-multiply.

j is "instruction pointer":
0 if at top of loop,
1 if in middle of loop.

Each "instruction" here includes exactly one multiply.

Try to choose instruction set with big useful operations, avoiding control overhead.

Analogous to designing CPU.

approach, not well known:

```
256bit(x,e):
 = 1,255,0
 i >= 0:
j == 0:
 = y*y
f 1&(e>>i):
 j = 1
lse:
 i = i-1
e:
 = y*x
,j = i-1,0
n y
```


This is like CPU's perspective on original square-and-multiply.

j is "instruction pointer":
0 if at top of loop,
1 if in middle of loop.

Each "instruction" here includes exactly one multiply.

Try to choose instruction set with big useful operations, avoiding control overhead.

Analogous to designing CPU.

Followin
assumin
i shifts e
assumin

```
def pow
  y,i,j
  while
    z =
    y =
    bit
    i =
    j =
  retur
```

, not well known:

```
e):
0
):
0
```


This is like CPU's perspective on original square-and-multiply.

j is "instruction pointer":
0 if at top of loop,
1 if in middle of loop.

Each "instruction" here includes exactly one multiply.

Try to choose instruction set with big useful operations, avoiding control overhead.

Analogous to designing CPU.

Following data-flo
assuming all arithr
i shifts etc.) is cor
assuming *e* weight

```
def pow256bit(x,
  y,i,j = 1,255,
  while i >= 0:
    z = y+(x-y)*
    y = y*z
    bit = 1&(e>>
    i = i-(j|(1-
    j = bit&(1-j
  return y
```

known:

This is like CPU's perspective on original square-and-multiply.

j is "instruction pointer": 0 if at top of loop, 1 if in middle of loop.

Each "instruction" here includes exactly one multiply.

Try to choose instruction set with big useful operations, avoiding control overhead.

Analogous to designing CPU.

Following data-flow rules, assuming all arithmetic (incl
i shifts etc.) is constant-tim
assuming e weight exactly 1

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    z = y+(x-y)*j
    y = y*z
    bit = 1&(e>>i)
    i = i-(j|(1-bit))
    j = bit&(1-j)
  return y
```

This is like CPU's perspective on original square-and-multiply.

j is "instruction pointer":
0 if at top of loop,
1 if in middle of loop.

Each "instruction" here includes exactly one multiply.

Try to choose instruction set with big useful operations, avoiding control overhead.

Analogous to designing CPU.

Following data-flow rules, assuming all arithmetic (including i shifts etc.) is constant-time, assuming e weight exactly 128:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    z = y+(x-y)*j
    y = y*z
    bit = 1&(e>>i)
    i = i-(j|(1-bit))
    j = bit&(1-j)
  return y
```

ike CPU's perspective on
square-and-multiply.

truction pointer":
op of loop,
middle of loop.

nstruction" here
exactly one multiply.

hoose instruction set
useful operations,
control overhead.

us to designing CPU.

Following data-flow rules, assuming all arithmetic (including i shifts etc.) is constant-time, assuming e weight exactly 128:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    z = y+(x-y)*j
    y = y*z
    bit = 1&(e>>i)
    i = i-(j|(1-bit))
    j = bit&(1-j)
  return y
```

Allowing

```
def pow
  y,i,j
  for l
    z =
    z =
    y =
    bit
    i =
    j =
  asser
  retur
```

perspective on
d-multiply.

ointer”:

,
oop.

” here
he multiply.

ruction set
erations,
verhead.

gning CPU.

Following data-flow rules,
assuming all arithmetic (including
i shifts etc.) is constant-time,
assuming e weight exactly 128:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    z = y+(x-y)*j
    y = y*z
    bit = 1&(e>>i)
    i = i-(j|(1-bit))
    j = bit&(1-j)
  return y
```

Allowing any weig

```
def pow256bitwei
  y,i,j = 1,255,
  for loop in ra
    z = y+(x-y)*
    z = z+(1-z)*
    y = y*z
    bit = 1&(e>>
    i = i-(j|(1-
    j = bit&(1-j
  assert i < 0
  return y
```

ve on

y.

t

J.

Following data-flow rules, assuming all arithmetic (including i shifts etc.) is constant-time, assuming e weight exactly 128:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    z = y+(x-y)*j
    y = y*z
    bit = 1&(e>>i)
    i = i-(j|(1-bit))
    j = bit&(1-j)
  return y
```

Allowing any weight $\leq$128:

```
def pow256bitweightle128(
  y,i,j = 1,255,0
  for loop in range(384):
    z = y+(x-y)*j
    z = z+(1-z)*(i<0)
    y = y*z
    bit = 1&(e>>max(i,0))
    i = i-(j|(1-bit))
    j = bit&(1-j)
  assert i < 0
  return y
```

Following data-flow rules, assuming all arithmetic (including i shifts etc.) is constant-time, assuming e weight exactly 128:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    z = y+(x-y)*j
    y = y*z
    bit = 1&(e>>i)
    i = i-(j|(1-bit))
    j = bit&(1-j)
  return y
```


Allowing any weight $\leq$128:

```
def pow256bitweightle128(x,e):
  y,i,j = 1,255,0
  for loop in range(384):
    z = y+(x-y)*j
    z = z+(1-z)*(i<0)
    y = y*z
    bit = 1&(e>>max(i,0))
    i = i-(j|(1-bit))
    j = bit&(1-j)
  assert i < 0
  return y
```

Following data-flow rules, assuming all arithmetic (including i shifts etc.) is constant-time, assuming e weight exactly 128:

```
def pow256bit(x,e):
  y,i,j = 1,255,0
  while i >= 0:
    z = y+(x-y)*j
    y = y*z
    bit = 1&(e>>i)
    i = i-(j|(1-bit))
    j = bit&(1-j)
  return y
```


Allowing any weight $\leq$128:

```
def pow256bitweightle128(x,e):
  y,i,j = 1,255,0
  for loop in range(384):
    z = y+(x-y)*j
    z = z+(1-z)*(i<0)
    y = y*z
    bit = 1&(e>>max(i,0))
    i = i-(j|(1-bit))
    j = bit&(1-j)
  assert i < 0
  return y
```

Exercise: constant-time ECC scalar mult with sliding windows.