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

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.

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.
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.

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.
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.

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.
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”):

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

ip ("instruction pointer" or "program counter"): address in RAM of next instruction.
Hardware reality: Accessing RAM is inherently expensive.

CPU designers try to reduce cost. Example: “L1 cache” typically has 32KB of recently used data. It inspects RAM addresses, performs various computations on addresses to save time.

Time is a function of RAM accesses. 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”):

```python
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.

```python
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.”
3 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.

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"):

```python
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.

```python
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".
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”):

```python
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.

```python
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.”
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”):

```python
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.

```python
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”.
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.

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”.

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
```
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.

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

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

Another approach, not well known:

```python
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
```
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”):**

```python
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.

```python
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:**

```python
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
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.
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

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:

```python
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.
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.
Another approach, not well known:

```python
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.

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

```python
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.
Another approach, not well known:

```python
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.

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

```python
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
```

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

Analogous to designing CPU.
Another approach, not well known:

```python
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.

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

```python
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:

```python
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 square-and-multiply.

“Instruction pointer”: top of loop, middle of loop.

“Instruction” here exactly one multiply.

Choose instruction set useful operations, control overhead.

Alas to designing CPU.

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

```python
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 $\leq 128$:

```python
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
```
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:

```python
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 \(\leq 128\):

```python
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
```
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:

```python
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 \( \leq 128 \):

```python
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:

```python
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 \( \leq 128 \):

```python
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:

```python
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 $\leq 128$:

```python
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.