Hash-based signatures I

Hash functions and one-time signatures

Daniel J. Bernstein[1,2,4] and Tanja Lange[3,4]

[1]University of Illinois at Chicago

[2]Ruhr University Bochum

[3]Eindhoven University of Technology

[4]Academia Sinica

1 April 2022

Python snippets for this talk:
https://cr.yp.to/talks/2022.04.01-2/ots-20220401.tar.gz

The SHA-256 cryptographic hash function

```
$ echo hello
hello
$
```

The SHA-256 cryptographic hash function

```
$ echo hello
hello
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$
```

The SHA-256 cryptographic hash function

```
$ echo hello
hello
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$ echo world | sha256sum
e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101eb317  -
$
```

The SHA-256 cryptographic hash function

```
$ echo hello
hello
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$ echo world | sha256sum
e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101eb317  -
$ echo this is a longer message | sha256sum
c316678498bdf2a77d64e1f3af0cdc6e943234d19ce38034e24ccf98a5ab5901  -
$
```

The SHA-256 cryptographic hash function

```
$ echo hello
hello
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$ echo world | sha256sum
e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101eb317  -
$ echo this is a longer message | sha256sum
c316678498bdf2a77d64e1f3af0cdc6e943234d19ce38034e24ccf98a5ab5901  -
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$
```

The SHA-256 cryptographic hash function

```
$ echo hello
hello
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$ echo world | sha256sum
e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101eb317  -
$ echo this is a longer message | sha256sum
c316678498bdf2a77d64e1f3af0cdc6e943234d19ce38034e24ccf98a5ab5901  -
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$
```

The `sha256sum` program computes the SHA-256 hash function.
This is a function $H : \{0,1\}^* \to \{0,1\}^{256}$. Each output is 32 bytes.

The SHA-256 cryptographic hash function

```
$ echo hello
hello
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$ echo world | sha256sum
e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101eb317  -
$ echo this is a longer message | sha256sum
c316678498bdf2a77d64e1f3af0cdc6e943234d19ce38034e24ccf98a5ab5901  -
$ echo hello | sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
$
```

The `sha256sum` program computes the SHA-256 hash function.
This is a function $H : \{0,1\}^* \to \{0,1\}^{256}$. Each output is 32 bytes.

(Actually, SHA-256 requires input to be at most $2^{64} - 1$ bits.
Exercise: Compute $\#$ years for today's fastest CPU to reach this limit.)

The SHA-256 cryptographic hash function in Python 3

```
>>> import hashlib
>>> def sha256(x):
...   h = hashlib.sha256()
...   h.update(x)
...   return h.digest()
...
>>> print(sha256(b'hello').hex())
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
>>>
```

The SHA-256 cryptographic hash function in Python 3

```
>>> import hashlib
>>> def sha256(x):
...   h = hashlib.sha256()
...   h.update(x)
...   return h.digest()
...
>>> print(sha256(b'hello').hex())
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
>>> print(sha256(b'hello\n').hex())
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03
>>>
```

The SHA-256 cryptographic hash function in Python 3

```
>>> import hashlib
>>> def sha256(x):
...   h = hashlib.sha256()
...   h.update(x)
...   return h.digest()
...
>>> print(sha256(b'hello').hex())
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
>>> print(sha256(b'hello\n').hex())
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03
>>> print(sha256(b'hello\n'*1000000).hex())
1a2cce61984891495b00826ef591104a34ff35766bbbcaaff965f766154812ab
>>>
```

Goals of cryptographic hash functions

What do we want from a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$?

For any string x, think of $H(x)$ as an n-bit fingerprint of x.

Goals:

- $H(x)$ looks totally random;
- nobody can find two different strings x, x' with $H(x) = H(x')$;
- any tiny change from x to x' makes a totally new $H(x')$;
- nobody can compute $H(x)$ without knowing all of x;
- nobody can compute a secret x given only $H(x)$;
- ...

Warning: Some hash goals are difficult to mathematically define.

Generic hardness of preimage resistance

Goal: Given $y \in H(\{0,1\}^*)$, finding $x \in \{0,1\}^*$ with $H(x) = y$ is hard.

Here y is given, and is known to be the image of some $x \in \{0,1\}^*$.
Typically there are many such x,
but it should be computationally hard to find any.

Generic hardness of preimage resistance

Goal: Given $y \in H(\{0,1\}^*)$, finding $x \in \{0,1\}^*$ with $H(x) = y$ is hard.

Here y is given, and is known to be the image of some $x \in \{0,1\}^*$.
Typically there are many such x,
but it should be computationally hard to find any.

Generic attack: Try $\approx 2^n$ random choices of x.
If the output of H is distributed uniformly then
each x has a $1/2^n$ chance of $H(x) = y$.

e.g. $\approx 2^{128}$ tries if $n = 128$: very expensive.

Generic hardness of preimage resistance

Goal: Given $y \in H(\{0,1\}^*)$, finding $x \in \{0,1\}^*$ with $H(x) = y$ is hard.

Here y is given, and is known to be the image of some $x \in \{0,1\}^*$.
Typically there are many such x,
but it should be computationally hard to find any.

Generic attack: Try $\approx 2^n$ random choices of x.
If the output of H is distributed uniformly then
each x has a $1/2^n$ chance of $H(x) = y$.

e.g. $\approx 2^{128}$ tries if $n = 128$: very expensive.

Exercise: Given $y_1, y_2, \ldots, y_{2^{20}}$,
how long does it take to find $x_1, x_2, \ldots, x_{2^{20}}$
such that $H(x_1) = y_1$ and $H(x_2) = y_2$ and $\ldots$ and $H(x_{2^{20}}) = y_{2^{20}}$?

Generic hardness of second-preimage resistance

Goal: Given $x \in \{0,1\}^*$, finding $x' \in \{0,1\}^*$ with $x \neq x'$ and $H(x') = H(x)$ is hard.

Here x is given, determining $y = H(x)$.
Typically there are many other $x' \neq x$ with the same image, but it should be computationally hard to find any.

Generic hardness of second-preimage resistance

Goal: Given $x \in \{0,1\}^*$, finding $x' \in \{0,1\}^*$ with $x \neq x'$ and $H(x') = H(x)$ is hard.

Here x is given, determining $y = H(x)$.
Typically there are many other $x' \neq x$ with the same image, but it should be computationally hard to find any.

Generic attack: Try $\approx 2^n$ random choices of $x' \neq x$.
Same speed as for first preimages.

Generic hardness of collision resistance

Goal: Finding $x, x' \in \{0,1\}^*$ with $x \neq x'$ and $H(x') = H(x)$ is hard.

Attacker has full flexibility to choose any output y.
It should still be computationally hard
to find two different strings x, x' with the same output.

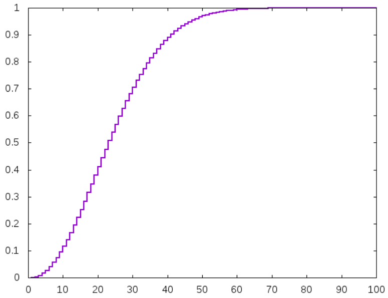
Generic hardness of collision resistance

Goal: Finding $x, x' \in \{0,1\}^*$ with $x \neq x'$ and $H(x') = H(x)$ is hard.

Attacker has full flexibility to choose any output y.
It should still be computationally hard
to find two different strings x, x' with the same output.

Generic attack: Try $\approx 2^{n/2}$ random choices of x. This number is much lower than the other two because there is no restriction on the target.

The "birthday paradox": if one draws $\approx 1.17\sqrt{m}$ elements at random from a set of m elements, then with 50% probability one has picked one element twice.



Weaknesses in common cryptographic hash functions

Some hash functions take n to be too small.
Some hash functions take n large enough
but have structure allowing faster attacks than the generic attacks.

Weaknesses in common cryptographic hash functions

Some hash functions take n to be too small.
Some hash functions take n large enough
but have structure allowing faster attacks than the generic attacks.

MD4 (1990 Rivest): $n = 128$, so 2^{64} generic collision attack.
MD4-specific collision attack (1995) in seconds.
Current best collision attack (2007) is even faster.

Weaknesses in common cryptographic hash functions

Some hash functions take n to be too small.
Some hash functions take n large enough
but have structure allowing faster attacks than the generic attacks.

MD4 (1990 Rivest): $n = 128$, so 2^{64} generic collision attack.
MD4-specific collision attack (1995) in seconds.
Current best collision attack (2007) is even faster.

MD5 (1992 Rivest): $n = 128$, so 2^{64} generic collision attack.
MD5-specific collision attack (2004) in one hour on a cluster.
Current best collision attack (2013) costs 2^{18} H calls.
Chosen-prefix collisions (2008) showed real-world exploitability.
Flame malware (2012) used MD5 collision to sign fake Windows update.

Weaknesses in common cryptographic hash functions

Some hash functions take n to be too small.
Some hash functions take n large enough
but have structure allowing faster attacks than the generic attacks.

MD4 (1990 Rivest): $n = 128$, so 2^{64} generic collision attack.
MD4-specific collision attack (1995) in seconds.
Current best collision attack (2007) is even faster.

MD5 (1992 Rivest): $n = 128$, so 2^{64} generic collision attack.
MD5-specific collision attack (2004) in one hour on a cluster.
Current best collision attack (2013) costs 2^{18} H calls.
Chosen-prefix collisions (2008) showed real-world exploitability.
Flame malware (2012) used MD5 collision to sign fake Windows update.

SHA-0 (1993 NSA): $n = 160$, so 2^{80} generic collision attack.
Many weaknesses found. Collisions published (2004).

Weaknesses in common cryptographic hash functions

Some hash functions take n to be too small.
Some hash functions take n large enough
but have structure allowing faster attacks than the generic attacks.

MD4 (1990 Rivest): $n = 128$, so 2^{64} generic collision attack.
MD4-specific collision attack (1995) in seconds.
Current best collision attack (2007) is even faster.

MD5 (1992 Rivest): $n = 128$, so 2^{64} generic collision attack.
MD5-specific collision attack (2004) in one hour on a cluster.
Current best collision attack (2013) costs 2^{18} H calls.
Chosen-prefix collisions (2008) showed real-world exploitability.
Flame malware (2012) used MD5 collision to sign fake Windows update.

SHA-0 (1993 NSA): $n = 160$, so 2^{80} generic collision attack.
Many weaknesses found. Collisions published (2004).

SHA-1 (1995 NSA): $n = 160$, so 2^{80} generic collision attack.
Collisions published (2017): https://shattered.io/.
Practical attack, chosen-prefix collision (2020):
https://sha-mbles.github.io/

The NSA view of cryptographic standardization

"Narrowing the encryption problem to a single, influential algorithm might drive out competitors, and that would reduce the field that NSA had to be concerned about. Could a public encryption standard be made secure enough to protect against everything but a massive brute force attack, but **weak enough to still permit an attack of some nature** using very sophisticated (and expensive) techniques?"

(Emphasis added.)

This quote is from an internal NSA history book.

Some unbroken hash functions

SHA-256 (2001 NSA): $n = 256$, so 2^{128} generic collision attack.

SHA-512 (2001 NSA): $n = 512$.

"SHA-2" refers to SHA-256, SHA-512, etc.

Some unbroken hash functions

SHA-256 (2001 NSA): $n = 256$, so 2^{128} generic collision attack.

SHA-512 (2001 NSA): $n = 512$.

"SHA-2" refers to SHA-256, SHA-512, etc.

2004–2005: Big improvements in attacks against MD5, SHA-1.
2007–2012: NIST holds "SHA-3" competition in case SHA-2 is broken.

Some unbroken hash functions

SHA-256 (2001 NSA): $n = 256$, so 2^{128} generic collision attack.

SHA-512 (2001 NSA): $n = 512$.

"SHA-2" refers to SHA-256, SHA-512, etc.

2004–2005: Big improvements in attacks against MD5, SHA-1.
2007–2012: NIST holds "SHA-3" competition in case SHA-2 is broken.

SHA3-256 (2015 Bertoni–Daemen–Peeters–van Assche): $n = 256$.

SHA3-512 (2015 Bertoni–Daemen–Peeters–van Assche): $n = 512$.

Some unbroken hash functions

SHA-256 (2001 NSA): $n = 256$, so 2^{128} generic collision attack.

SHA-512 (2001 NSA): $n = 512$.

"SHA-2" refers to SHA-256, SHA-512, etc.

2004–2005: Big improvements in attacks against MD5, SHA-1.
2007–2012: NIST holds "SHA-3" competition in case SHA-2 is broken.

SHA3-256 (2015 Bertoni–Daemen–Peeters–van Assche): $n = 256$.

SHA3-512 (2015 Bertoni–Daemen–Peeters–van Assche): $n = 512$.

Another popular SHA-3 finalist, faster than SHA-3 in software: BLAKE.

Hash-based signatures

Use a hash function to build a **public-key signature system**.
Old idea, starting with 1979 Lamport one-time signatures.
Many further improvements in years since.

Hash-based signatures

Use a hash function to build a **public-key signature system**.
Old idea, starting with 1979 Lamport one-time signatures.
Many further improvements in years since.

Signer generates secret key and public key.
Everyone learns signer's public key.

Using secret key, signer can sign any message m,
producing a signed message (m, s).

Everyone can verify (m, s) using signer's public key.

Hash-based signatures

Use a hash function to build a **public-key signature system**.
Old idea, starting with 1979 Lamport one-time signatures.
Many further improvements in years since.

Signer generates secret key and public key.
Everyone learns signer's public key.

Using secret key, signer can sign any message m,
producing a signed message (m, s).

Everyone can verify (m, s) using signer's public key.

Attacker's goal: construct (m, s) that signer didn't sign
but that passes verification using signer's public key.

Hash-based signatures

Use a hash function to build a **public-key signature system**.
Old idea, starting with 1979 Lamport one-time signatures.
Many further improvements in years since.

Signer generates secret key and public key.
Everyone learns signer's public key.

Using secret key, signer can sign any message m,
producing a signed message (m, s).

Everyone can verify (m, s) using signer's public key.

Attacker's goal: construct (m, s) that signer didn't sign
but that passes verification using signer's public key.

Attacker looks at public key and at signed messages.
Tries modifying the signed messages or creating new messages.

A signature scheme for empty messages: key generation

A signature scheme for empty messages: key generation

```
import os,hashlib

def sha3_256(x):
  h = hashlib.sha3_256()
  h.update(x)
  return h.digest()

def keypair():
  secret = sha3_256(os.urandom(32))
  public = sha3_256(secret)
  return public,secret
```

A signature scheme for empty messages: key generation

```
import os,hashlib

def sha3_256(x):
  h = hashlib.sha3_256()
  h.update(x)
  return h.digest()

def keypair():
  secret = sha3_256(os.urandom(32))
  public = sha3_256(secret)
  return public,secret
```

```
>>> import signempty
>>> pk,sk = signempty.keypair()
>>> pk.hex()
'61ba682f03259a276dc2d790ed4863113d5559ad7cdd3c282083b9aa6b170ff
>>> sk.hex()
'4645dd39db47dd18b646a34b8f2dc6afd7fa62cc6faafc2ad3426dc94394335
```

Signing and verifying empty messages

```
def sign(message,secret):
  if not isinstance(message,bytes):
    raise TypeError('message must be a byte string')
  if message != b'':
    raise ValueError('message must be empty')
  signedmessage = secret
  return signedmessage

def open(signedmessage,public):
  if len(signedmessage) != 32:
    raise ValueError('bad signature')
  if sha3_256(signedmessage) != public:
    raise ValueError('bad signature')
  message = b''
  return message
```

Signing and verifying empty messages

```
def sign(message,secret):
  if not isinstance(message,bytes):
    raise TypeError('message must be a byte string')
  if message != b'':
    raise ValueError('message must be empty')
  signedmessage = secret
  return signedmessage

def open(signedmessage,public):
  if len(signedmessage) != 32:
    raise ValueError('bad signature')
  if sha3_256(signedmessage) != public:
    raise ValueError('bad signature')
  message = b''
  return message
```

```
>>> sm = signempty.sign(b'',sk)
>>> signempty.open(sm,pk)
b''
```

A signature scheme for 1-bit messages: keygen, signing

```
import signempty

def keypair():
  p0,s0 = signempty.keypair()
  p1,s1 = signempty.keypair()
  return (p0,p1),(s0,s1)

def sign(message,secret):
  if not isinstance(message,bytes):
    raise TypeError('message must be a byte string')
  if message == b'0':
    return message,signempty.sign(b'',secret[0])
  if message == b'1':
    return message,signempty.sign(b'',secret[1])
  raise ValueError("message must be b'0' or b'1'")
```

A signature scheme for 1-bit messages: verification

```
def open(signedmessage,public):
  if not isinstance(signedmessage[0],bytes):
    raise TypeError('message must be a byte string')
  if signedmessage[0] == b'0':
    signempty.open(signedmessage[1],public[0])
    return b'0'
  if signedmessage[0] == b'1':
    signempty.open(signedmessage[1],public[1])
    return b'1'
  raise ValueError('bad signature')
```

A signature scheme for 1-bit messages: verification

```
def open(signedmessage,public):
  if not isinstance(signedmessage[0],bytes):
    raise TypeError('message must be a byte string')
  if signedmessage[0] == b'0':
    signempty.open(signedmessage[1],public[0])
    return b'0'
  if signedmessage[0] == b'1':
    signempty.open(signedmessage[1],public[1])
    return b'1'
  raise ValueError('bad signature')
```

```
>>> import signbit
>>> pk,sk = signbit.keypair()
>>> sm = signbit.sign(b'1',sk)
>>> signbit.open(sm,pk)
b'1'
```

A signature scheme for 4-bit messages: key generation

```
import signbit

def keypair():
  p0,s0 = signbit.keypair()
  p1,s1 = signbit.keypair()
  p2,s2 = signbit.keypair()
  p3,s3 = signbit.keypair()
  return (p0,p1,p2,p3),(s0,s1,s2,s3)

def sign(m,secret):
  if not isinstance(m,bytes):
    raise TypeError('message must be a byte string')
  if len(m) != 4:
    raise ValueError('message must have length 4')
  sm0 = signbit.sign(m[0:1],secret[0])
  sm1 = signbit.sign(m[1:2],secret[1])
  sm2 = signbit.sign(m[2:3],secret[2])
  sm3 = signbit.sign(m[3:4],secret[3])
  return sm0,sm1,sm2,sm3
```

A signature scheme for 4-bit messages: sign & verify

```
def open(sm,public):
  if len(sm) != 4:
    raise ValueError('signed message must have length 4')
  m0 = signbit.open(sm[0],public[0])
  m1 = signbit.open(sm[1],public[1])
  m2 = signbit.open(sm[2],public[2])
  m3 = signbit.open(sm[3],public[3])
  return m0+m1+m2+m3
```

Do not use one secret key to sign two messages!

```
>>> import sign4bits
>>> pk,sk = sign4bits.keypair()
>>> sm0111 = sign4bits.sign(b'0111',sk)
>>> sign4bits.open(sm0111,pk)
b'0111'
>>> sm1101 = sign4bits.sign(b'1101',sk)
>>> sign4bits.open(sm1101,pk)
b'1101'
```

Do not use one secret key to sign two messages!

```
>>> import sign4bits
>>> pk,sk = sign4bits.keypair()
>>> sm0111 = sign4bits.sign(b'0111',sk)
>>> sign4bits.open(sm0111,pk)
b'0111'
>>> sm1101 = sign4bits.sign(b'1101',sk)
>>> sign4bits.open(sm1101,pk)
b'1101'

>>> forgery = sm1101[:2]+sm0111[2:]
>>> sign4bits.open(forgery,pk)
b'1111'
```

Lamport's 1-time signature system

Sign arbitrary-length message by signing its 256-bit hash:

```
def hashbits(message):
  h = sha3_256(message)
  return [(b'0',b'1')[1&(h[i//8]>>(i%8))] for i in range(256)]

def keypair():
  keys = [signbit.keypair() for n in range(256)]
  return zip(*keys)

def sign(message,secret):
  hbits = hashbits(message)
  sigs = [signbit.sign(hbits[i],secret[i]) for i in range(256)]
  return sigs,message

def open(sm,public):
  if len(sm[0]) != 256:
    raise ValueError('wrong signature length')
  message = sm[1]
  hbits = hashbits(message)
  for i in range(256):
    if hbits[i] != signbit.open(sm[0][i],public[i]):
      raise ValueError('bit %d of hash does not match'%i)
  return message
```

Can we build shorter signatures?

How big are Lamport's signatures?

- Each Lamport signature has 256 `signbit` signatures.
- Each `signbit` signature has 1 `signempty` signature.
- Each `signempty` signature has one hash output (32 bytes).

Total 256 hash outputs (8192 bytes).

For a 4-bit message: 4 hash outputs (128 bytes).

Can we build shorter signatures?

How big are Lamport's signatures?

- Each Lamport signature has 256 `signbit` signatures.
- Each `signbit` signature has 1 `signempty` signature.
- Each `signempty` signature has one hash output (32 bytes).

Total 256 hash outputs (8192 bytes).

For a 4-bit message: 4 hash outputs (128 bytes).

Idea for doing better, just 1 hash output for a 4-bit message:

- Define
$$H^i(x) = H(H^{i-1}(x)) = \underbrace{H(H(\cdots(H}_{i \text{ times}}(x)))).$$
- Pick random sk, compute $\text{pk} = H^{16}(\text{sk})$.
- For message $m \in \{0, 1, \ldots, 15\}$ reveal $s = H^m(\text{sk})$ as signature.
- To verify check that $\text{pk} = H^{16-m}(s)$.

Can we build shorter signatures?

How big are Lamport's signatures?

- Each Lamport signature has 256 `signbit` signatures.
- Each `signbit` signature has 1 `signempty` signature.
- Each `signempty` signature has one hash output (32 bytes).

Total 256 hash outputs (8192 bytes).

For a 4-bit message: 4 hash outputs (128 bytes).

Idea for doing better, just 1 hash output for a 4-bit message:

- Define
$$H^i(x) = H(H^{i-1}(x)) = \underbrace{H(H(\cdots(H}_{i \text{ times}}(x)))).$$
- Pick random sk, compute $\mathrm{pk} = H^{16}(\mathrm{sk})$.
- For message $m \in \{0, 1, \ldots, 15\}$ reveal $s = H^m(\mathrm{sk})$ as signature.
- To verify check that $\mathrm{pk} = H^{16-m}(s)$.

This is the weak Winternitz signature system.

Weak Winternitz

```
def keypair():
  secret = sha3_256(os.urandom(32))
  public = secret
  for i in range(16): public = sha3_256(public)
  return public,secret

def sign(m,secret):
  if not isinstance(m,int) or m<0 or m>15:
    raise ValueError('message must be in {0,1,...,15}')
  s = secret
  for i in range(m): s = sha3_256(s)
  return s,m

def open(sm,public):
  if not isinstance(sm[1],int) or sm[1]<0 or sm[1]>15:
    raise ValueError('message must be in {0,1,...,15}')
  c = sm[0]
  for i in range(16-sm[1]): c = sha3_256(c)
  if c != public: raise ValueError('bad signature')
  return sm[1]
```

Why this is “weak” Winternitz

This is insecure even if you sign only 1 message!

```
>>> import weak_winternitz
>>> pk,sk = weak_winternitz.keypair()
>>> sm7 = weak_winternitz.sign(7,sk)
>>> H = weak_winternitz.sha3_256
>>> weak_winternitz.open(sm7,pk)
7
>>> forgery = H(sm7[0]),8
>>> weak_winternitz.open(forgery,pk)
8
>>> forgery2 = H(forgery[0]),9
>>> weak_winternitz.open(forgery2,pk)
9
>>>
```

Why this is “weak” Winternitz

This is insecure even if you sign only 1 message!

```
>>> import weak_winternitz
>>> pk,sk = weak_winternitz.keypair()
>>> sm7 = weak_winternitz.sign(7,sk)
>>> H = weak_winternitz.sha3_256
>>> weak_winternitz.open(sm7,pk)
7
>>> forgery = H(sm7[0]),8
>>> weak_winternitz.open(forgery,pk)
8
>>> forgery2 = H(forgery[0]),9
>>> weak_winternitz.open(forgery2,pk)
9
>>>
```

Fix: Strong Winternitz uses weak Winternitz twice,
running one chain forward, one chain backward.
(Exercise: this is safe with H^{15} instead of H^{16} in weak Winternitz.)

Strong Winternitz

```
import weak_winternitz

def keypair():
  keys = [weak_winternitz.keypair() for n in range(2)]
  return zip(*keys)

def sign(m,secret):
  if not isinstance(m,int) or m<0 or m>15:
    raise ValueError('message must be in {0,1,...,15}')
  sign0 = weak_winternitz.sign(m,secret[0])
  sign1 = weak_winternitz.sign(15-m,secret[1])
  return sign0[0],sign1[0],m

def open(sm,public):
  if not isinstance(sm[2],int) or sm[2]<0 or sm[2]>15:
    raise ValueError('message must be in {0,1,...,15}')
  weak_winternitz.open((sm[0],sm[2]),public[0])
  weak_winternitz.open((sm[1],15-sm[2]),public[1])
  return sm[2]
```

The complete Winternitz system

Define parameter w. Each chain will run for 2^w steps.

For signing a 256-bit hash this needs $t_1 = \lceil 256/w \rceil$ chains.
Write m in base 2^w (integers of w bits):

$$m = (m_{t_1-1}, \ldots, m_1, m_0)$$

(zero–padding if necessary).

Put

$$c = \sum_{i=0}^{t_1-1} (2^w - m_i)$$

Note that $c \leq t_1 2^w$.

The checksum c gets larger if m_i is smaller.

Write c in base 2^w. This takes $t_2 = 1 + \lceil \lfloor (\log_2 t_1 \rfloor + 1)/w \rceil$ w-bit integers

$$c = (c_{t_2-1}, \ldots, c_1, c_0).$$

Publish $t_1 + t_2$ public keys, sign with chains of lengths

$$m_{t_1-1}, \ldots, m_1, m_0, c_{t_2-1}, \ldots, c_1, c_0.$$

The complete Winternitz system for $w = 8$

Define parameter $w = 8$. Each chain will run for $2^8 = 256$ steps.

For signing a 256-bit hash this needs $t_1 = \lceil 256/8 \rceil = 32$ chains.
Write m in base 2^8 (integers of 8 bits):

$$m = (m_{31}, \ldots, m_1, m_0)$$

(zero–padding if necessary).

Put

$$c = \sum_{i=0}^{31} (2^8 - m_i)$$

Note that $c \leq 32 \cdot 2^8 = 2^{13}$.

The checksum c gets larger if m_i is smaller.

Write c in base 2^8. This takes $t_2 = 1 + \lceil (5+1)/8 \rceil = 2$
8-bit integers

$$c = (c_1, c_0).$$

Publish $t_1 + t_2 = 34$ public keys, sign with chains of lengths

$$m_{31}, \ldots, m_1, m_0, c_1, c_0.$$

Exercise

How does Winternitz with $w = 5$ compare to Winternitz with $w = 8$ for signing a 256-bit hash? Efficiency metrics:

- How many bytes are in the signature?
- How many bytes are in the public key?
- How many bytes are in the secret key?
- How many hash-function computations are needed in signing?
- How many hash-function computations are needed in verifying?

Remember that you also need to sign the checksum component!