Hash-based signatures I

Basic concepts

Tanja Lange
(with some slides by Daniel J. Bernstein)

Eindhoven University of Technology

SAC – Post-quantum cryptography

Benefits of hash-based signatures

- Old idea: 1979 Lamport one-time signatures.
- 1979 Merkle extends to more signatures;
  many further improvements in years since.
- Security thoroughly analyzed.
- Only one prerequisite: a good hash function, e.g. SHA3-512, ...
  Hash functions map long strings to fixed-length strings.

$$H : \{0,1\}^* \to \{0,1\}^n.$$

  Signature schemes use hash functions in handling m.
- Cryptographic hash functions are computationally
  - preimage resistant: function is one way;
  - second preimage resistant:
    given $x, H(x)$ cannot find $x' \neq x$ with $H(x') = H(x)$;
  - collision resistant: cannot find $x' \neq x$ with $H(x') = H(x)$.

  Quantum computers affect the hardness only marginally
  finding preimages in $2^{n/2}$ instead of 2^n (Grover, not Shor).

A signature scheme for empty messages: key generation

A signature scheme for empty messages: key generation

First part of signempty.py

```
import os; from hashlib import sha3_256;

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

A signature scheme for empty messages: key generation

First part of signempty.py

```
import os; from hashlib import sha3_256;

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

```
>>> import signempty; import binascii;
>>> pk,sk = signempty.keypair()
>>> binascii.hexlify(pk)
b'a447bc8d7c661f85defcf1bbf8bad77bfc6191068a8b658c99c7ef4cbe37cf
>>> binascii.hexlify(sk)
b'a4a1334a6926d04c4aa7cd98231f4b644be90303e4090c358f2946f1c25768
```

A signature scheme for empty messages: signing, verification

Rest of signempty.py

```
def sign(message,secret):
  if message != '': raise Exception('nonempty message')
  signedmessage = secret
  return signedmessage

def open(signedmessage,public):
  if sha3_256(signedmessage) != public:
    raise Exception('bad signature')
  message = ''
  return message
```

A signature scheme for empty messages: signing, verification

Rest of signempty.py

```
def sign(message,secret):
  if message != '': raise Exception('nonempty message')
  signedmessage = secret
  return signedmessage

def open(signedmessage,public):
  if sha3_256(signedmessage) != public:
    raise Exception('bad signature')
  message = ''
  return message
```

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

A signature scheme for 1-bit messages: key generation, signing

A signature scheme for 1-bit messages: key generation, signing

First part of signbit.py

```
import signempty

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

def sign(message,secret):
  if message == 0:
    return ('0' , signempty.sign('',secret[0:32]))
  if message == 1:
    return ('1' , signempty.sign('',secret[32:64]))
  raise Exception('message must be 0 or 1')
```

A signature scheme for 1-bit messages: verification

Rest of signbit.py

```
def open(signedmessage,public):
  if signedmessage[0] == '0':
    signempty.open(signedmessage[1],public[0:32])
    return 0
  if signedmessage[0] == '1':
    signempty.open(signedmessage[1],public[32:64])
    return 1
  raise Exception('message must be 0 or 1')
```

A signature scheme for 1-bit messages: verification

Rest of signbit.py

```
def open(signedmessage,public):
  if signedmessage[0] == '0':
    signempty.open(signedmessage[1],public[0:32])
    return 0
  if signedmessage[0] == '1':
    signempty.open(signedmessage[1],public[32:64])
    return 1
  raise Exception('message must be 0 or 1')
```

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

A signature scheme for 4-bit messages: key generation

First part of sign4bits.py

```
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
```

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

Rest of sign4bits.py

```
def sign(m,secret):
  if type(m) != int: raise Exception('message must be int')
  if m < 0 or m > 15:
    raise Exception('message must be between 0 and 15')
  sm0 = signbit.sign(1 & (m >> 0),secret[0:64])
  sm1 = signbit.sign(1 & (m >> 1),secret[64:128])
  sm2 = signbit.sign(1 & (m >> 2),secret[128:192])
  sm3 = signbit.sign(1 & (m >> 3),secret[192:256])
  return sm0+sm1+sm2+sm3

def open(sm,public):
  m0 = signbit.open(sm[0:2],public[0:64])
  m1 = signbit.open(sm[2:4],public[64:128])
  m2 = signbit.open(sm[4:6],public[128:192])
  m3 = signbit.open(sm[6:],public[192:256])
  return m0 + 2*m1 + 4*m2 + 8*m3
```

Do not use one secret key to sign two messages!

```
>>> import sign4bits
>>> pk,sk = sign4bits.keypair()
>>> sm11 = sign4bits.sign(11,sk)
>>> sign4bits.open(sm11,pk)
11
>>> sm7 = sign4bits.sign(7,sk)
>>> sign4bits.open(sm7,pk)
7
```

Do not use one secret key to sign two messages!

```
>>> import sign4bits
>>> pk,sk = sign4bits.keypair()
>>> sm11 = sign4bits.sign(11,sk)
>>> sign4bits.open(sm11,pk)
11
>>> sm7 = sign4bits.sign(7,sk)
>>> sign4bits.open(sm7,pk)
7

>>> forgery = sm7[:6] + sm11[6:]
>>> sign4bits.open(forgery,pk)
15
```