FINE 434: FinTech

Lecture 7

Professor Fahad Saleh

McGill University - Desautels

Introduction

A Hash function, $H : \mathcal{M} \mapsto \mathcal{T}$, constitutes a function with the following properties:

- $\mathcal{M}$, the "message space", is an infinite space
- $\mathcal{T}$, the "digest space", is a finite space
- "efficiently computable"

When we apply H to some $m \in \mathcal{M}$, we say that we have "hashed" m.

Motivating Application

Problem: You want to patent a “secret” but without revealing it because you worry about people stealing your idea.

Motivating Application

Problem: You want to patent a "secret" but without revealing it because you worry about people stealing your idea.

Solution: Take your secret, s, and provide $H(s)$ to the regulator. If somebody else produces idea s' then $s = s'$ implies $H(s) = H(s')$ so that you may verify an infringement on your patent despite not having revealed s.

Motivating Application

Problem: You want to patent a “secret” but without revealing it because you worry about people stealing your idea.

Solution: Take your secret, s, and provide $H(s)$ to the regulator. If somebody else produces idea s' then $s = s'$ implies $H(s) = H(s')$ so that you may verify an infringement on your patent despite not having revealed s.

... not so fast! What could go wrong? What properties do we need to overcome these issues?

Hash Function Security Properties

- Collision-Free
- Hiding
- Puzzle-Friendly

Collision-Free

$s' = s$ implies $H(s) = H(s')$ but what about the converse?

What if somebody else develops $s' \neq s$ but $H(s') = H(s)$? For example, what if $H(m) = 0$ for all $m \in \mathcal{M}$?

Collision-Free

$s' = s$ implies $H(s) = H(s')$ but what about the converse?

What if somebody else develops $s' \neq s$ but $H(s') = H(s)$? For example, what if $H(m) = 0$ for all $m \in \mathcal{M}$?

We want H to be one-to-one (i.e. $m = m' \Leftrightarrow H(m) = H(m')$)

Collision-Free

$s' = s$ implies $H(s) = H(s')$ but what about the converse?

What if somebody else develops $s' \neq s$ but $H(s') = H(s)$? For example, what if $H(m) = 0$ for all $m \in \mathcal{M}$?

We want H to be one-to-one (i.e. $m = m' \Leftrightarrow H(m) = H(m')$)
... but that's not possible. Why?

Collision-Free

$s' = s$ implies $H(s) = H(s')$ but what about the converse?

What if somebody else develops $s' \neq s$ but $H(s') = H(s)$? For example, what if $H(m) = 0$ for all $m \in \mathcal{M}$?

We want H to be one-to-one (i.e. $m = m' \Leftrightarrow H(m) = H(m')$)
... but that's not possible. Why?
Input space is larger than output space!

A hash function is collision-free if it is impractical to find two inputs, m and m', such that $H(m) = H(m')$.

SHATTERED

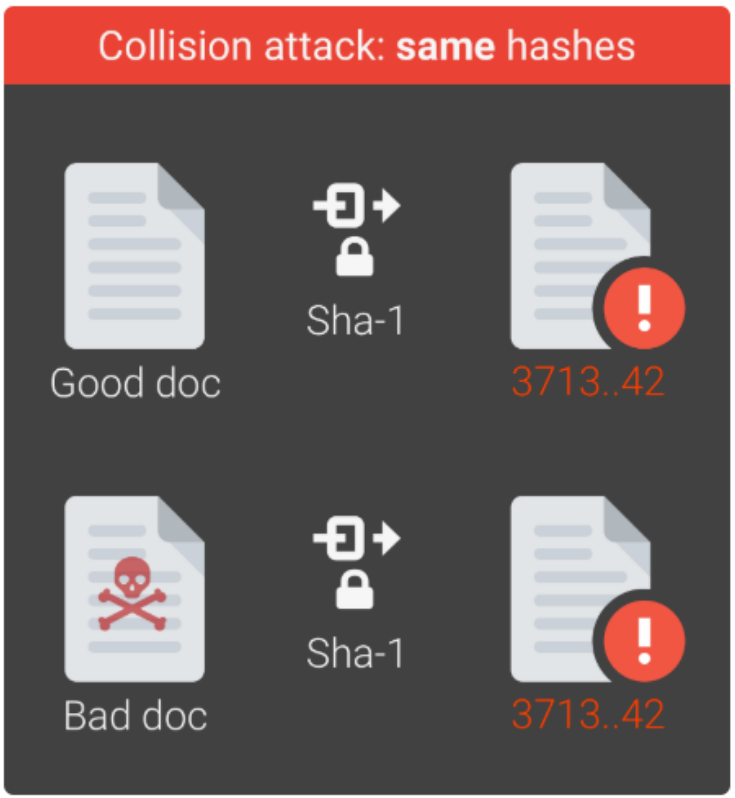
We have broken SHA-1 in practice.

This industry cryptographic hash function standard is used for digital signatures and file integrity verification, and protects a wide spectrum of digital assets, including credit card transactions, electronic documents, open-source software repositories and software updates.

It is now practically possible to craft two colliding PDF files and obtain a SHA-1 digital signature on the first PDF file which can also be abused as a valid signature on the second PDF file.

For example, by crafting the two colliding PDF files as two rental agreements with different rent, it is possible to trick someone to create a valid signature for a high-rent contract by having him or her sign a low-rent contract.

Infographic | Paper



Hiding

Assume $\mathcal{M} = \mathbb{N}$ and $\mathcal{T} = \{0, 1, ..., 2^{256} - 1\}$. Let $H(m) = m$ for $m \in \mathcal{T}$. What's wrong (in terms of our patent example)?

Hiding

Assume $\mathcal{M} = \mathbb{N}$ and $\mathcal{T} = \{0, 1, ..., 2^{256} - 1\}$. Let $H(m) = m$ for $m \in \mathcal{T}$. What's wrong (in terms of our patent example)?

It is easy to find m such that $H(m) = H(s)$! This is either a collision or $m = s$ and the procedure provides no secrecy.

To provide secrecy, we need the hiding property.

Informally, H has the hiding property if it is difficult to decipher m when knowing only $H(m)$

Hiding

Assume $\mathcal{M} = \mathbb{N}$ and $\mathcal{T} = \{0, 1, ..., 2^{256} - 1\}$. Let $H(m) = m$ for $m \in \mathcal{T}$. What's wrong (in terms of our patent example)?

It is easy to find m such that $H(m) = H(s)$! This is either a collision or $m = s$ and the procedure provides no secrecy.

To provide secrecy, we need the hiding property.

Informally, H has the hiding property if it is difficult to decipher m when knowing only $H(m)$

Actually, hiding involves difficulty of deciphering m if hashed jointly with some (sufficiently random) noise. (Why?)

Puzzle-Friendly

Informally: Given $t \in \mathcal{T}$, it is difficult to find $m \in \mathcal{M}$ such that m hashes to t when hashed with some (sufficiently random) noise.

i.e.
There is no solving strategy better than brute force.

The relevance of this property may not be clear now, but it is a crucial property for Bitcoin mining.

hashlib

Python contains a package, hashlib, that possesses various hashing functions. Bitcoin employs SHA256, so we restrict our attention to SHA256 and SHA1.

To use hashlib, type import hashlib at the top of your code.

```
1 import hashlib
2
```

encode(...)

Both SHA1 and SHA256 require inputs as bytes, so we need to convert our text into bytes before applying either hash function.

encode(...) is a str function, so we must precede encode by a str object and “.”

```
1 text = "this is a string object"
2 encodedtext = text.encode()
3 print(type(encodedtext))
```

```
<class 'bytes'>
```

SHA1

The method for SHA1 is sha1. The method, however, is from the hashlib class, so we need to run the command hashlib.sha1(encodedtext)

```
1 import hashlib
2
3 text = "this is a message"
4 encodedtext = text.encode()
5 testhash = hashlib.sha1(encodedtext)
6 print(type(testhash))
```

```
<class '_hashlib.HASH'>
```

hexdigest

hashlib.sha1(...) returns a complicated “object.” To get a readable output of the hash, we need to apply a function, hexdigest, to get the hash output.

```
1 import hashlib
2
3 text = "this is a message"
4 encodedtext = text.encode()
5 testhash = hashlib.sha1(encodedtext)
6
7 print(testhash.hexdigest())
```

```
d9cd24603a4903704c8e2f887c92e1663b2a737e
```

digest

Alternatively, to get the output as bytes, we need to apply a function, digest.

```
1  import hashlib
2
3  text = "this is a message"
4  encodedtext = text.encode()
5  testhash = hashlib.sha1(encodedtext)
6
7  print(testhash.digest())
8
9  print(type(testhash.digest()))
```

```
b'\xd9\xcd$`:I\x03pL\x8e/\x88|\x92\xe1f;*s~'
<class 'bytes'>
```

SHA256

```python
import hashlib

text = "this is a message"
encodedtext = text.encode()
testhash = hashlib.sha256(encodedtext)

print(testhash.hexdigest())
```

```
cee86e2a6c441f1e308d16a3db20a8fa8fae2a45730b48ca2c0c61e159af7e78
```

SHA256

```python
import hashlib

text = "this is a message"
encodedtext = text.encode()
testhash = hashlib.sha256(encodedtext)
hexdigest = testhash.hexdigest()

print(int(hexdigest,16))
```

```
93587115662999119233973122137893007199961142348250461256530355637180685057656
```

Introduction

We conventionally use base 10 (“decimal”). In decimal, a number’s representation depends upon the powers of 10.

e.g.
$523 = \mathbf{5} \times 10^2 + \mathbf{2} \times 10^1 + \mathbf{3} \times 10^0$
$29 = \mathbf{2} \times 10^1 + \mathbf{9} \times 10^0$

Introduction

We conventionally use base 10 ("decimal"). In decimal, a number's representation depends upon the powers of 10.

e.g.
$523 = \mathbf{5} \times 10^2 + \mathbf{2} \times 10^1 + \mathbf{3} \times 10^0$
$29 = \mathbf{2} \times 10^1 + \mathbf{9} \times 10^0$

In general, the decimal representation for $x \in \mathbb{N}$, is $a_n a_{n-1} ... a_0$ if $x = \sum_{i=0}^{n} a_i 10^i$ and $\forall i : a_i \in \{0, ..., 9\}$

Binary

Typically, computers store information in base 2 (“binary”). In binary, a number’s representation depends upon the powers of 2.

e.g.
$2 = 10_2 = \mathbf{1} \times 2^1 + \mathbf{0} \times 2^0$
$129 = 10000001_2 = \mathbf{1} \times 2^7 + \mathbf{0} \times 2^6 + ... + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0$

In general, the binary representation for $x \in \mathbb{N}$, is $a_n a_{n-1} ... a_0$ if $x = \sum_{i=0}^{n} a_i 2^i$ and $\forall i : a_i \in \{0, 1\}$

Hexidecimal

Note that 129 possesses 8 digits in binary. Binary can be lengthy, so a computer may represent binary information in a base 16 ("hexidecimal") format.

e.g.

$16 = 10_{16} = \mathbf{1} \times 16^1 + \mathbf{0} \times 16^0$

$270 = 10(14)_{16} = \mathbf{1} \times 16^2 + \mathbf{0} \times 16^1 + \mathbf{14} \times 16^0$

Hexidecimal

Note that 129 possesses 8 digits in binary. Binary can be lengthy, so a computer may represent binary information in a base 16 ("hexidecimal") format.

e.g.

$16 = 10_{16} = \mathbf{1} \times 16^1 + \mathbf{0} \times 16^0$

$270 = 10(14)_{16} = \mathbf{1} \times 16^2 + \mathbf{0} \times 16^1 + \mathbf{14} \times 16^0$

Having a "two-digit" number as a single digit is not ideal...

Hexidecimal

Note that 129 possesses 8 digits in binary. Binary can be lengthy, so a computer may represent binary information in a base 16 ("hexidecimal") format.

e.g.
$16 = 10_{16} = \mathbf{1} \times 16^1 + \mathbf{0} \times 16^0$
$270 = 10(14)_{16} = \mathbf{1} \times 16^2 + \mathbf{0} \times 16^1 + \mathbf{14} \times 16^0$

Having a "two-digit" number as a single digit is not ideal...
... but our notion of "two-digit" numbers comes from decimal (... that's why 10 is the first two-digit number) and is arbitrary, so we can simply expand the set of one-digit characters to overcome this issue.

Number	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal	0	1	2	3	4	5	6	7

Number	8	9	10	11	12	13	14	15
Binary	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal	8	9	A	B	C	D	E	F

In general, the hex representation for $x \in \mathbb{N}$, is $a_n a_{n-1} ... a_0$ if $x = \sum_{i=0}^{n} a_i 16^i$ and $\forall i : a_i \in \{0, ..., 15\}$

Loose Ends

In general, the base $b \in \mathbb{N}$ representation for $x \in \mathbb{N}$, is $a_n a_{n-1} ... a_0$ if $x = \sum_{i=0}^{n} a_i b^i$ and $\forall i : a_i \in \{0, ..., b-1\}$

.hexdigest() gives a hexidecimal representation of binary information (... which we can and will interpret as a number when we study blockchain)

SHA256 produces a longer hex sequence than SHA1 because SHA256's output corresponds to 256 binary digits ("bits"). To how many binary digits does SHA1's output correspond?

Loose Ends

In general, the base $b \in \mathbb{N}$ representation for $x \in \mathbb{N}$, is $a_n a_{n-1} ... a_0$ if $x = \sum_{i=0}^{n} a_i b^i$ and $\forall i : a_i \in \{0, ..., b-1\}$

.hexdigest() gives a hexidecimal representation of binary information (... which we can and will interpret as a number when we study blockchain)

SHA256 produces a longer hex sequence than SHA1 because SHA256's output corresponds to 256 binary digits ("bits"). To how many binary digits does SHA1's output correspond? 160.

There are 40 characters in the output. Each character is hex and therefore corresponds to 4 binary digits. ($2^4 = 16$)