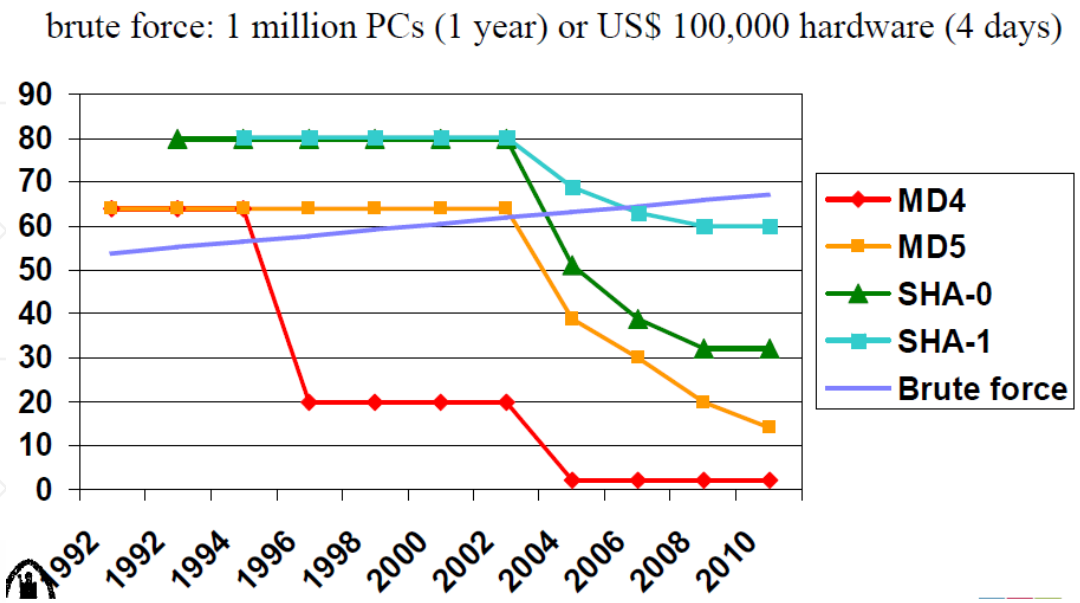
**Symposium on Distributed Ledger Technology SDLT'2017**
Gold Coast Campus, Griffith University
June 13, 2017




# Design Patterns which Facilitate Message Digest Collision Attacks on Blockchains

Peter Robinson, peter.robinson@sent.com
School of Information Technology and Electrical Engineering, University of Queensland.

# Overview

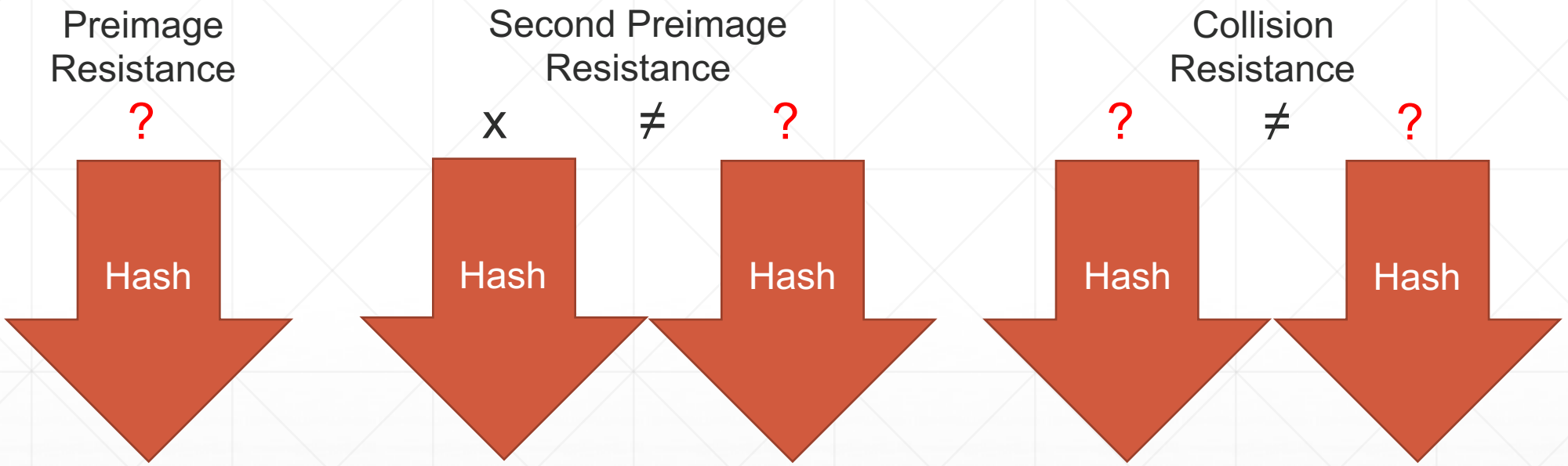
- Message digest algorithms are one of the underlying building blocks of blockchain platforms such as Ethereum.
- This paper analyses situations in which the message digest collision resistance property can be exploited by attackers.
- Two mitigations for possible attacks are described:
  - Longer message digest sizes make attacks more difficult; and,
  - Including timeliness properties limits the amount of time an attacker has to determine a hash collision.

**Ref 1**: Image on cover slide: Preneel, B. (2013) "Introduction to the Design and Cryptanalysis of Cryptographic Hash Functions" Available: https://www.cosic.esat.kuleuven.be/summer_school_albena/slides/preneel_hash_july2013_shortv1_print.pdf

# Message Digest / Cryptographic Hash Algorithms

- Digest Algorithm (Hash):
  - Variable length input => Fixed Length Output.
- Address Hash:
  - Bitcoin and Ripple: RIPEMD160(SHA256): 160 bit digest,
  - Ethereum: KECCAK/160: 160 bit digest.
- Main Hash:
  - Bitcoin: SHA256(SHA256): 256 bit digest,
  - Ethereum: KECCAK/256: 256 bit digest,
  - Ripple: 256 bit truncated SHA512: 256 bit digest.

# Message Digest Security Strength & Cryptanalysis




| | | | |
|---|---|---|---|
| Classical | n | n | n/2 |
| Quantum | n/2 | n/2 | n/3 |

# NIST Recommendation on Security Strength

- 80 bit Security Strength algorithms should not be used after 2010[2].
- 160 bit message digest functions provide 80 bits of security strength for collision resistance property.
- Address hashes used by Bitcoin, Ethereum, and Ripple may be susceptible to collision attacks by large state based actors.

**Ref 2**: NIST (2009) "DISCUSSION PAPER: The Transitioning of Cryptographic Algorithms and Key Sizes"
http://csrc.nist.gov/groups/ST/key_mgmt/documents/Transitioning_CryptoAlgos_070209.pdf
SP800-57: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf
EINSA: https://www.enisa.europa.eu/publications/algorithms-key-sizes-and.../fullReport

# Design Patterns to Avoid / Exploitation Scenarios

- The attacker chooses the value to be digested.
- The attacker is able to trick an entity into doing something based the message digest and value.
- There is no time-sensitive aspect to the value.
  - Time-sensitivity is important as the first message digest collisions for an algorithm have historically taken a significant period of time.

# Attack Scenario #1: EIP86 Transaction Verification Contracts

- Contract is deployed the address calculated as:

  Keccak256(rlp:encode([creator+nonce+initcode]))%$2^{160}$

- Where:
  - **creator** is the account that created the contract,
  - **nonce** is a value which should be sequential,
  - **initcode** is the initialization code for the contract.

```
# Get signature from tx data
sig_v = ~calldataload(0)
sig_r = ~calldataload(32)
sig_s = ~calldataload(64)
# Get tx arguments
tx_nonce = ~calldataload(96)
tx_to = ~calldataload(128)
tx_value = ~calldataload(160)
tx_gasprice = ~calldataload(192)
tx_data = string(~calldatasize() - 224)
~calldataload(tx_data, 224, ~calldatasize())
# Get signing hash
s_data = string(~calldatasize() - 64)
~mstore(s_data, tx.startgas)
~calldataload(s_data+32, 96, ~calldatasize()-96)
s_hash = sha3(s_data:str)
# Perform usual checks
prev_nonce = ~sload(-1)
assert tx_nonce == prev_nonce + 1
assert self.balance >=
    tx_value + tx_gasprice * tx.startgas
assert ~ecrecover(s_hash, sig_v, sig_r, sig_s)
    == <pubkey hash here>
# Update nonce
~sstore(-1, prev_nonce + 1)
# Pay for gas
~send(MINER_CONTRACT, tx_gasprice * tx.startgas)
# Make the main call
~call(msg.gas - 50000, tx_to, tx_value, tx_data,
    len(tx_data), 0, 0)
# Get remaining gas payments back
~call(20000, MINER_CONTRACT,0,[msg.gas],32,0,0)
```

# Attack Scenario #1: EIP86 Transaction Verification Contracts

- An attacker who had broken the message digest collision resistance property could manipulate the **nonce** value or the contents of the contract such that there exists two contracts which would be deployed to the same address.
- The attacker could show one variant of the contract to a user and persuade them to send some Ether to the address. The attacker could then deploy the nefarious contract and spend the Ether.

```
# Get signature from tx data
sig_v = ~calldataload(0)
sig_r = ~calldataload(32)
sig_s = ~calldataload(64)
# Get tx arguments
tx_nonce = ~calldataload(96)
tx_to = ~calldataload(128)
tx_value = ~calldataload(160)
tx_gasprice = ~calldataload(192)
tx_data = string(~calldatasize() - 224)
~calldataload(tx_data, 224, ~calldatasize())
# Get signing hash
s_data = string(~calldatasize() - 64)
~mstore(s_data, tx.startgas)
~calldataload(s_data+32,96,~calldatasize()-96)
s_hash = sha3(s_data:str)
# Perform usual checks
prev_nonce = ~sload(-1)
assert tx_nonce == prev_nonce + 1
assert self.balance >=
    tx_value + tx_gasprice * tx.startgas
assert ~ecrecover(s_hash, sig_v, sig_r, sig_s)
    == <pubkey hash here>
# Update nonce
~sstore(-1, prev_nonce + 1)
# Pay for gas
~send(MINER_CONTRACT, tx_gasprice * tx.startgas)
# Make the main call
~call(msg.gas - 50000, tx_to, tx_value, tx_data,
    len(tx_data), 0, 0)
# Get remaining gas payments back
~call(20000, MINER_CONTRACT,0,[msg.gas],32,0,0)
```

# Attack Scenario #2: Proposal for Proof of Stake Agreed Random Number

- Ethereum's Proof of Stake proposal needs an agreed random number to use to choose which miner will create the next block.
- One proposal is:
  - Each miner generates a random value,
  - To prevent cheating, miners submit commitment values: message digest(random value),
  - The random values are then exposed and combined using XOR to produce the agreed random number.

# Attack Scenario #2: Proposal for Proof of Stake Agreed Random Number

- An attacker who had broken the collision resistance property could:
  - Determine two random values which hash to produce the same commitment value,
  - Submit their commitment value,
  - Then, after seeing the random values exposed by the other miners, choose which of their random values to deliver, thus influencing the resulting agreed random number.

# Design Pattern & Attack Mitigations

- The message digest size used for account numbers should be increased.
  - In an email from V. Buterin and M. Swende from the Ethereum team on May 18, 2017, they tentatively agreed to adopt this recommendation of mine.
- A temporal component should be incorporated into designs.
  - The random value proposed for use in PoS should incorporate some value from a recent block.

# Questions