Program Analysis for System Security and Reliability (Spring 2020)

Summary

Author:

Yannick Merkli

8 Pages

Contents

1 Introduction to blockchains and smart contracts

1.1 Background

1.1.1 Hash functions and crypto puzzles

Hash functions are functions that take an arbitrary-sized input and map it to a fixed-size output.
Hash functions are further:

- Deterministic (equal inputs hash to the same value)
- Uniform (inputs are mapped evenly to the space of possible outputs)
- Efficient (should be fast to compute)

Since we have $|input\_space| >> |output\_space|$ (the input space is inherently unbound), collisions exist. However, cryptographic hash functions ensure collisions are hard to find.

Cryptographic hash functions
Consider a hash function $h : X \to Y$:

- Pre-image resistant: given $y \in Y$, it is infeasible to find $x \in X$ such that $h(x) = y$
- Second pre-image resistant: given $x \in X$, it is infeasible to find $x' \in X$ such that $x \neq x'$ and $h(x) = h(x')$
- Collision resistance: It is infeasible to find a pair $(x, x') \in X \times X$ such that $x \neq x'$ and $h(x) \neq h(x')$. Collision resistance implies second pre-image resistance.

Applications of cryptographic hash functions:

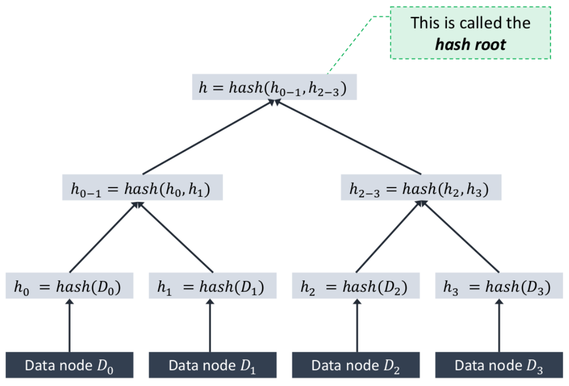
- Data equality: If we know that $h(x) = h(y)$ and h is a cryptographic hash function, then it is safe to assume that $x = y$ (due to collision resistance).
- Data integrity:
  - To verify the integrity a piece of data d, we can remember its hash, $hash = h(d)$
  - Later, when we obtain d' from untrusted source, we can verify whether $hash = h(d')$, to check whether $d = d'$
  - Useful because $hash$ is small (e.g. 256 bits)

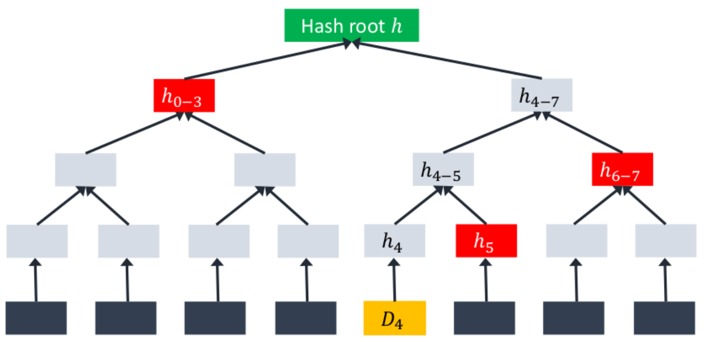
  Example: blockchain, don't need to remember the whole chain, just a hash of it
- Crypto puzzles:
  - Puzzle-friendly: For any output y, if r is chosen from a probability distribution with high min-entropy (true random), then it is infeasible to find x such that $h(r|x) = y$
  - Search puzzle: Given a puzzle ID id, chosen from a probability distribution with high min-entropy, and an output target $T \subset Y$, find a solution x such that $h(id|x) \in T$. Puzzle-friendliness implies that no solving strategy is much better than trying random values of x (i.e. brute-force).

  Example: Bitcoin proof-of-work.

1.1.2 Merkle trees

A Merkle tree is built bottom up. Start by hashing all the data and then recursively build the tree by hashing the hashes of the two children which becomes the parent hash.





Data integrity verification: The **hash root** h is typically obtained from a trusted source. One can verify the integrity of the data elements D_0, D_1, D_2, D_3 by reconstructing the hash root and comparing it to the trusted root.

Verifying whether D_4 is a member (i.e., a leaf): Obtain the hash root h from a trusted source. Then request the nodes h_5, h_{6-7}, h_{0-3} (from possibly untrusted source). Finally, compute $h_4, h_{4-5}, h_{0-3}, h'$ and check

whether $h = h'$.
Membership verification requires $\log(n)$ elements, only *one* of which needs to be from a trusted resource (the hash root h). This is especially useful when the set of data elements is large.
Possible application: check if blockchain transaction is valid (i.e. in the tree).

1.1.3 Digital signatures

The goal of a digital signature is to allow only one user to sign but anyone to verify the signature. We have the following API for signatures:

- $(sk, pk) = generateKeys(keySize)$
  - sk is the secret key, which the owner needs to keep private
  - pk is the public key, which is distributed to all users
- $sig = sign(sk, msg)$
- $verify(pk, msg, sig)$

Establish an identity:

- User generates (sk, pk) key pair
- $hash(pk)$ is the public name of the user
- sk allows the user to endorse a statement $stmt$ using a digital signature $sig = sign(sk, smt)$
- anyone can verify statements endorsed by the user using $verify(pk, stmt, sig)$

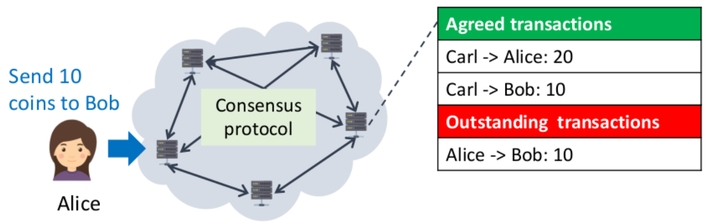
1.2 Bitcoin

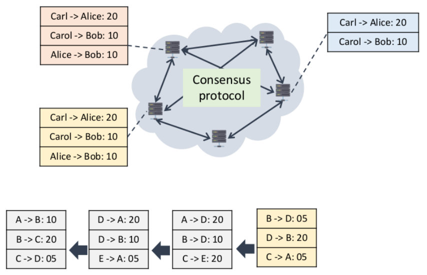
In a traditional financial system, we have a centralized ledger that keeps track of everyone's balance (i.e. the amount of money). This would essentially be a very big table. In cryptocurrencies such as Bitcoin, we have a *distributed ledger*. In a distributed ledger, we don't have a central table with all balances. There are multiple balances in the network. However, all nodes need to agree on the state of the ledger. For this, the ledgers run a distributed consensus protocol. Distributed consensus needs to fulfill:

- Termination: Every correct process decides on some value
- Agreement: All correct processes decide on the same value
- Validity: If all correct processes propose the same value, then any correct process must decide on that value

Traditional motivation: reliability in distributed systems (node replication).

To make a transaction, users broadcast transactions to the network nodes. All nodes have a sequence of all blocks of agreed transactions they have reached consensus on. Each node has a set of outstanding transactions (to be added to a block in the blockchain). Distributed consensus in necessary since all nodes need to agree on the set of accepted transactions. Otherwise, problems such as double-spending arise.

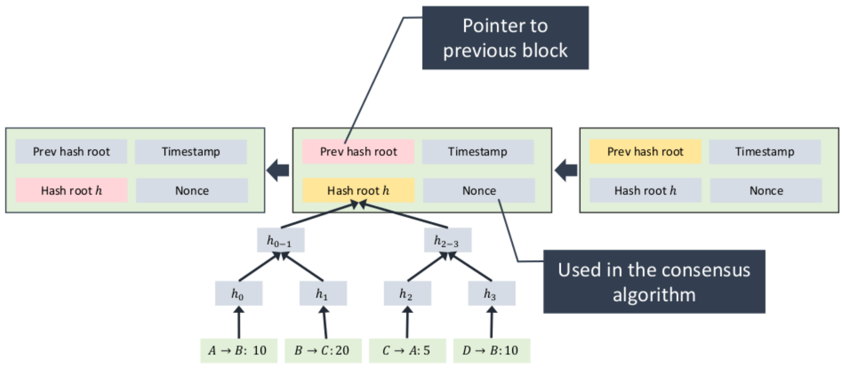




The blockchain itself as a datastrucure is essentially a linked list of blocks. A block contains a set of accepted transactions. More precisely, a block contains:

- Prev hash root: Merkle hash root of the last block
- Hash root h: Merkle hash root of the current block. The Merkle hash tree contains all transactions included in the current block. The Merkle tree enables record integrity of all transaction and efficient membership verification.
- Timestamp
- Nonce

Distributed consensus is hard: nodes may crash, nodes may be byzantine, network is imperfect (not all pairs of nodes are connected, messages may have arbitrary delays, no global time).



1.2.1 Consensus algorithm (simplified)

1. New transactions are broadcast to all nodes
2. Each node collects new transactions into a block
3. In each round, a **random node** gets to broadcast its block
4. Other nodes accept the block only if all transactions in it are valid (unspent, valid signatures)
5. Nodes express their acceptance of the block by including its hash in the next block they create

The question is how to select a random node. This selection needs to be careful since if we were to select nodes without randomness, it could be that nodes belonging to the same entity pick the transactions repeatedly which would for example allow the entity to censor transactions. Several schemes exist on how to pick a node: proof-of-work (select in proportion to computing power), proof-of-stake (select in proportion to ownership),...

1.2.2 Proof of work in Bitcoin

Nodes need to solve a hard puzzle. Nodes have the previous block with its hash root h_{prev} and the Merkle tree of the current block (with root h), consisting of all new transactions. The nodes then need to find a nonce X such that:

$$hash(h_{prev}|h|X) \leq difficulty$$

Changing the parameter $difficulty$ adjusts the expected time it takes until a node finds a nonce X (i.e. lower $difficulty$ means it takes longer). The current difficulty of Bitcoin is 15,546,745,765,529, which results in a average rate of 10.2min per block. The current hash power is 103,559,611,798 GH/s.
Proof-of-work makes it easy to verify: other nodes verify $hash(h_{prev}|h_{tx}) \leq difficulty$.
Key security assumption: Attacks are infeasible if majority of miners weighted by hash power follow the protocol.

1.2.3 Double spending

If the same output is spent in two different transactions, we have a double-spending. This results in a fork of the blockchain. Honest nodes always extend the longest valid branch. Over time, shorter branches will be abandoned. There are incentives for nodes to act honest and extend the longest chain:

- Block reward: The creator of a block can include a special coin-creation transaction in the block which will send a fixed amount of bitcoin (halfed every 210'000 blocks, currently 12.5BTC) to an chosen recipient address of this transaction. Block creator "receives" the reward only if the block ends up on the long-term consensus branch.
- Transaction fees: Creator of transaction may choose to make output value less than input value: $\sum inputs \geq \sum outputs$. Remainder ($\sum inputs - \sum outputs$) is transaction fee that goes to the block creator.
  By increasing the transaction fees, you can control how fast your transaction is processed (a higher transaction fee gives a bigger incentive for miners to include the transaction in the block).

1.2.4 Transactions

Each transaction output is in one of two possible states: spent or unspent. The sum of all unspent transaction outputs (UTXOs) builds part of the shared state of the blockchain. A transaction is valid iff $\sum inputs \geq \sum outputs$.

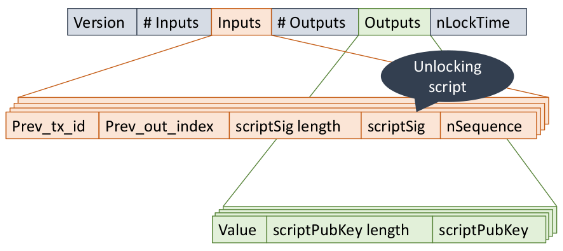


Figure 3: Bitcoin transaction structure

1.2.5 Bitcoin scripting

A transaction can contain a special script. On the most basic level, we have an unlocking script: this controls who can spend an unspent transaction output, which gives ownership. The sender uses the recipient's public key as parameter for `CHECKSIG`. The recipient then provides a signature generated with its private key (which can only be done by the holder of the private key), which is then checked by `CHECKSIG`.
Common Bitcoin scripts include: P2PK (Payment to public key), P2PKH (Payment to public key hash), P2MS (Pay to multi-signatures) (need multiple keys to unlock).

There also exists a stronger version of Bitcoin scripts: BitML - a language for Bitcoin smart contracts. A possible application is a timed commitment:

- Alice wants to commit to a secret s, but reveal it sometime later.
- Bob wants to be assured that he will either: learn Alice's secret s within time t or be compensated after time t.
- To achieve this, Alice can chooes and broadcasts $h = H(S)$ and releases two transactions:
  - (reveal h. withdraw A): Alice can reveal s and spend the coin herself (i.e. send back to herself)
  - (after t: withdraw B): Bob can spend the coin after time t

Bitcoin scripts are limited though: no loops, no signature verification on arbitrary messages, no multiplication and shifting, no arithmetic on long numbers, no concatenation on bitstrings. Bitcoin scripting is not Turing complete.

2 Smart contracts and Ethereum

A smart contract formalizes the terms of a contract in code.

A smart contract is a **computerized transaction protocol** that executes the **terms of a contract**. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and **minimize the need for trusted intermediaries**. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.
Traditional vs smart contracts:

	Traditional	Smart
Specification	Natural language	Code
Identity & consent	Signatures	Digital signatures
Dispute resolution	Judges, arbitrators	Decentralized platform
Nullification	By judges	???
Payment	As specified	built-in
Escrow	Trusted third party	built-in

Judges are not needed in smart contracts for dispute resolution since smart contracts are precise and dont allow for interpretation (as language does). Further, nullification is not possible since once a smart contract is released, it lives on its own and isn't owned by anyone.

2.1 Ethereum

A decentralized platform designed to run smart contracts:

- Similar to a world computer that executes code and maintains the state of all smart contracts
- The latest block stores the latest local states of all smart contracts
- Transactions result in executing code (calling a function) in target smart contracts
- Transaction change the state of one more more contracts
- Ethereum smart contracts are Turing-complete

Once a contract is released it lives on its own and might be processed at some point by miners.

Bitcoin vs Ethereum

- Bitcoin: Bob owns a private key to a set of unspent transactions (UTXOs). The sum of all UTXOs which can be unlocked with Bob's private key is Bob's Bitcoin balance. This makes it easy to make transactions and to prevent double-spending attacks.
- Ethereum: Bob owns private keys to an account. An account has an address, a balance and some executable code. To update the balance, we update the balance of the account instead of storing unspent transactions.

Ethereum accounts

- User account: These are owned by some external entity (person, corporation,...). User accounts can send transactions to transfer Ether or trigger contract code. Contains: address, Ether balance.
- Contract account: These are 'owned' by a contract (autonomous). But there is no real ownership for released contracts. Code execution is triggered by transactions that call functions. Contains: address, Ether balance, Associated contract code, Persistent storage (state).

Smart contracts are written in some high-level language (Solidity, Vyper). The high-level language is then compiled to a single low-level language. Currently, the low-level language is EVM (Ethereum virtual machine) bytecode. But this is to be replaced by Ethereum WebAssembly (eWASM) in Ethereum 2.0.
Once a smart contract is deployed to Ethereum, it is immutable and exists forever. If you write a bug, you don't get to take down the broken version; you can only fix-forward [1].

Example smart contract:

```
pragma solidity 0.5.8; //specify Solidity version
contract SimpleBank { //contract name
  mapping(address => uint) balances;
  function deposit(uint amount) payable public {
    balances[msg.sender] += amount;
  }
  function withdraw() public {
    msg.sender.transfer(balances[msg.sender]);
    balances[msg.sender] = 0;
  }
}
```

with:

- mapping(address => uint) balances: Local state maintains a mapping from user addresses to unsigned integers
- deposit(uint amount): Function that allows users to deposit ether at the SimpleBank contract
- payable: Function can receive Ether
- public: Any user can invoke the function
- msg.sender returns the address of the transaction sender
- withdraw(): Trigger a transfer of ETH with value equal to balances[msg.sender] to the transaction sender and then set the balance of the transaction sender to zero.

[1]there's a selfdestruct() function that a contract can use to remove itself from the network

2.1.1 Ethereum transactions

From	<address>	Address of the transaction sender
To	<address>	Address of the target contract
Data	<method>, <arg>,...,<arg>	ID of the method to invoke, along with arguments
Value	<amount>	Amount of Ether sent to the target contract
Gas/ gas price		

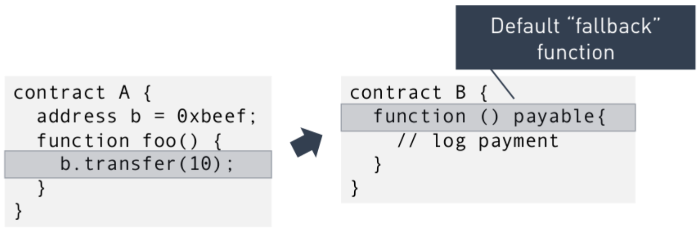
Gas
A transaction could contain an infinite loop (Ethereum is Turing complete) and thus a miner would need to process it forever. The solution to this is *gas*:

- A transaction requires "gas" to fuel contract execution
- Each EVM opcode requires gas to execute
- Some EVM opcodes consume variable amount of gas (e.g. SSTORE)
- Every transaction specifies the maximum ether the sender is willing to spent on the transaction (max gas + gas price)
- If the contract successfully executes, the unspent ether is refunded to the sender
- If execution runs out of gas, the execution reverts without refunding ether to the transaction sender

Thus, in case of an infinite loop, the transaction will eventually run out of gas and the execution will be reverted.
Different EVM opcodes require different amounts of gas. Some operations even cost a negative amount of gas: If you use storage in the blockchain (using `SSTORE`) and then free it up, you get a reward → negative gas price.

An attacker can exploit the gas mechanism by crafting a transaction that triggers an out-of-gas exception at an arbitrary point in the execution. This may create undesirable behaviors if the out-of-gas exception does not revert the entire transaction.

Ether transfers
An ether transfer to a contract implicitly calls the receiver contract. Contract B takes control over the execution:



```
contract A {
  address b = 0xbeef;
  function foo() {
    b.transfer(10);
  }
}
```

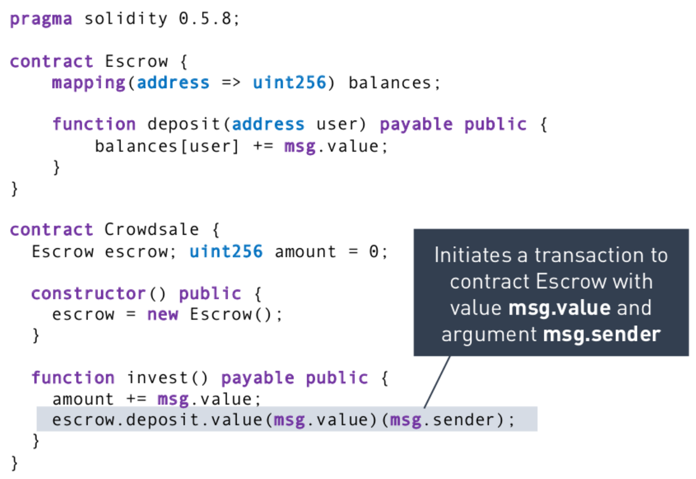
```
contract B {
  function () payable{
    // log payment
  }
}
```

Figure 4: Ether transfers

In order to avoid target contracts exploiting this, a Gas limit which is provide to the receiver was put into place. Types of ether transfers:

	Gas provided to the receiver	Error propagation
transfer(amount)	2300	yes
send(amount)	2300	no
call.value(amount)()	All remaining gas	no

The gas provided to the receiver can also be manually specified, e.g. `.value(amount).gas(1000)`



```
pragma solidity 0.5.8;

contract Escrow {
    mapping(address => uint256) balances;

    function deposit(address user) payable public {
        balances[user] += msg.value;
    }
}

contract Crowdsale {
  Escrow escrow; uint256 amount = 0;

  constructor() public {
    escrow = new Escrow();
  }

  function invest() payable public {
    amount += msg.value;
    escrow.deposit.value(msg.value)(msg.sender);
  }
}
```

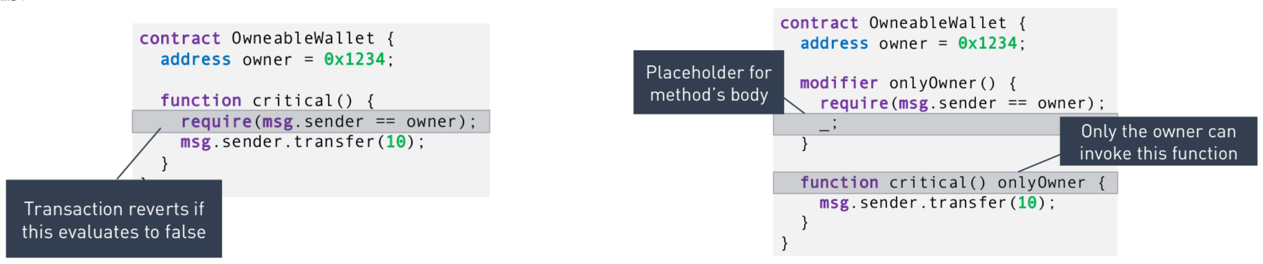
Initiates a transaction to contract Escrow with value **msg.value** and argument **msg.sender**

2.1.2 Contract calls

How can an attacker exploit calls to external contracts? The attacker may own the external contract and can thus execute arbitrary code. In particular, the attacker may "call back" the contract before returning control to the caller (we will see an example of this vulnerability later).

2.1.3 Authorization in smart contracts

Any user can call arbitrary functions in contracts. The contract must explicitly restrict access to sensitive functions.



```
contract OwneableWallet {
  address owner = 0x1234;

  function critical() {
    require(msg.sender == owner);
    msg.sender.transfer(10);
  }
}
```

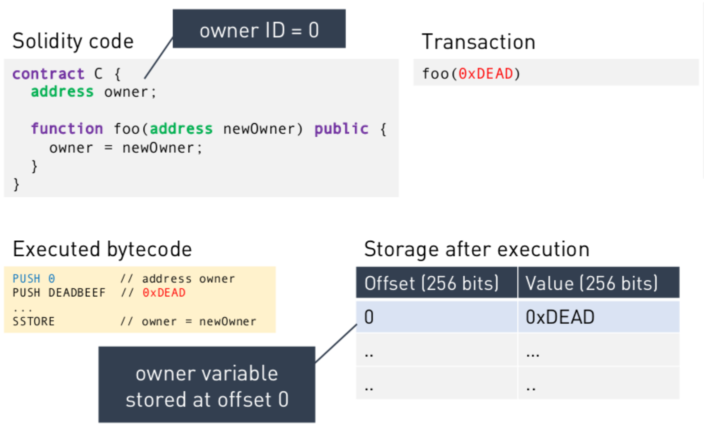
```
contract OwneableWallet {
  address owner = 0x1234;

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function critical() onlyOwner {
    msg.sender.transfer(10);
  }
}
```

Transaction reverts if this evaluates to false
Placeholder for method's body
Only the owner can invoke this function

2.1.4 Storage model

The storage model of Ethereum is essentially one huge array. There is no isolation in the storage model, everything goes everywhere! The storage array has 2^{256} entries (thus an offset of 256 bits) and a 256 bit wide value per entry. Storage is public since miners need to know it.

- **Variables** are stored at the offset specified by their index.
  Example: owner = 0xFeedBeef, Stored at offset = owner ID
- **Mappings** have a base offset which comes from their index. The offset of the values in the mappings are calculated through: $hash(mapping\_base||maping\_key)$.
  Example: balances[key] = 10, Offset = $SHA3(balancesID||key)$
- **Arrays** have a base offset which comes from a hash of their index. Arrays then linearly extend.
  Example: investors[10] = 0xFeed, Offset = $SHA3(arrID) + 10$



Solidity code

```
contract C {
  address owner;

  function foo(address newOwner) public {
    owner = newOwner;
  }
}
```

Transaction

```
foo(0xDEAD)
```

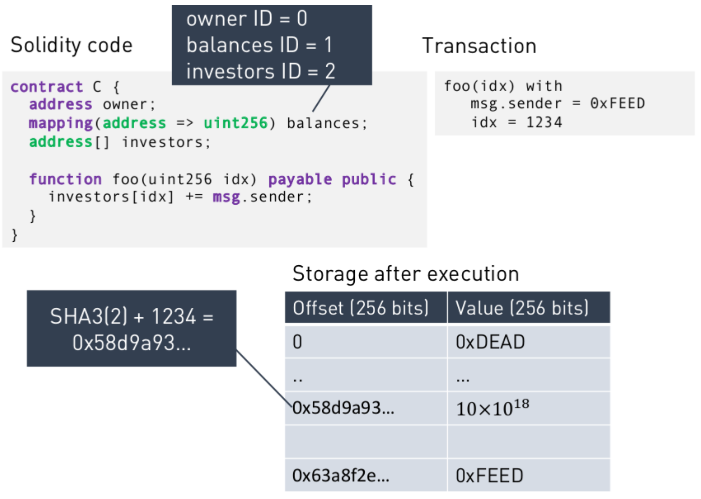
Executed bytecode

```
PUSH 0          // address owner
PUSH DEADBEEF   // 0xDEAD
...
SSTORE          // owner = newOwner
```

Storage after execution

Offset (256 bits)	Value (256 bits)
0	0xDEAD
..	...
..	..

(a) Storage model for variables



Solidity code

```
contract C {
  address owner;
  mapping(address => uint256) balances;
  address[] investors;

  function foo(uint256 idx) payable public {
    investors[idx] += msg.sender;
  }
}
```

Transaction

```
foo(idx) with
  msg.sender = 0xFEED
  idx = 1234
```

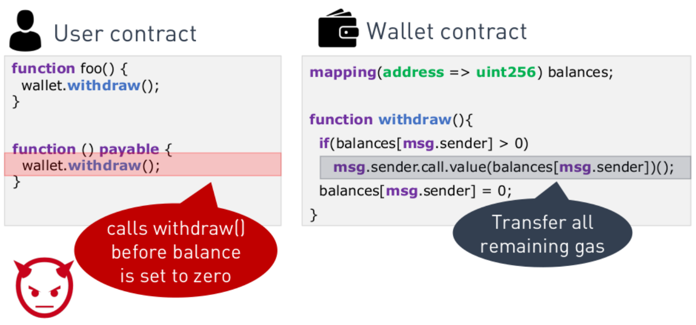
Storage after execution

Offset (256 bits)	Value (256 bits)
0	0xDEAD
..	...
0x58d9a93...	10×10^{18}
0x63a8f2e...	0xFEED

(b) Storage model for mappings

2.2 Vulnerabilities of smart contracts

2.2.1 The DAO bug

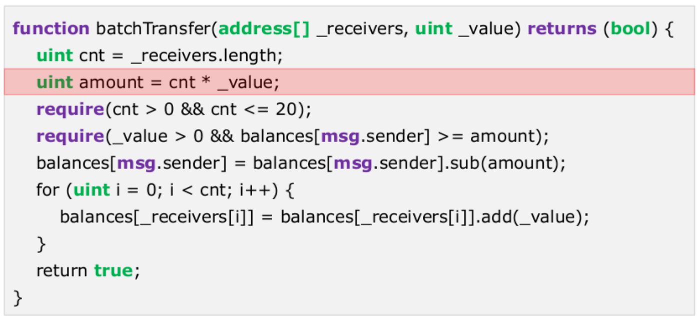

User contract
Wallet contract
calls withdraw() before balance is set to zero
Transfer all remaining gas

```
function foo() {
  wallet.withdraw();
}

function () payable {
  wallet.withdraw();
}
```

```
mapping(address => uint256) balances;

function withdraw(){
  if(balances[msg.sender] > 0)
    msg.sender.call.value(balances[msg.sender])();
  balances[msg.sender] = 0;
}
```

In the wallet contract code in the withdraw function, the balance is set to zero *after* the amount is sent to the msg.sender. The malicious sender would then call withdraw again before the balance is set to zero and could thus withdraw the same amount multiple times.

2.2.2 BatchTransfer ERC20 Bug



```
function batchTransfer(address[] _receivers, uint _value) returns (bool) {
    uint cnt = _receivers.length;
    uint amount = cnt * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);
    balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] = balances[_receivers[i]].add(_value);
    }
    return true;
}
```

An attacker can send a very large `_value` → all senders will get this very large value. However, the very large value will cause an overflow in `amount`, which results in `amount` becoming small. The sender will only be substracted `amount` (small)!

2.3 Semantics and security properties

References