A Concurrent Perspective on Smart Contracts

Ilya Sergey

Aquinas Hobor

1st Workshop on Trusted Smart Contracts
7 April 2017
class ConcurrentQueue <E> {
    public synchronized void enqueue(E elem) {...}
    public synchronized E dequeue() {...}
}
class ConcurrentQueue <E> {
    public synchronized void enqueue(E elem) {...}
    public synchronized E dequeue() {...}
}

class MyQClient {
    public void foo (ConcurrentQueue<Integer> q) {
        ...
        q.enqueue(1);
        q.enqueue(2);
        doStuff();
        Integer i = q.dequeue();
        assert (i == 1);
        q.dequeue();
    }
}
class MyQClient {
    public void foo (ConcurrentQueue<Integer> q) {
        ...
        q.enqueue(1);
        q.enqueue(2);
        doStuff();
        Integer i = q.dequeue();
        assert (i == 1);
        q.dequeue();
    }
}

Queue q = new ConcurrentQueue<Integer>();
MyQClient c1 = new MyQClient();
MyQClient c2 = new MyQClient();

c1.foo(q) || c2.foo(q)
```java
class MyQClient {
    public void foo (ConcurrentQueue<Integer> q) {
        ...
        q.enqueue(1);
        q.enqueue(2);
        doStuff();
        Integer i = q.dequeue();
        assert (i == 1);
        q.dequeue();
    }
}
```

The diagram shows two calls to `foo(q)` from different threads (`c1` and `c2`). The thread `c1` enqueues 1 and 2, and then dequeues 2, but the assertion `assert (i == 1)` fails because 2 was dequeued instead of 1. The thread `c2` also enqueues 1 and 2, but does not attempt to dequeue, and as such, the assertion is not applicable since there is no dequeue operation executed. The assertion failure indicates a race condition in the dequeuing process, as the order of enqueuing and dequeuing operations is not guaranteed in a concurrent queue environment.
**contract** MyQContract {  

    Queue q = QueueContract(0x1d11e5fbe221);

**function** foo() {  

    ...
    q.enqueue(\texttt{addr1});
    q.enqueue(\texttt{addr2});
    \texttt{someAddr}.call.value(...);
    address i = q.dequeue();
    // Assuming \( i == \texttt{addr1} \)
    i.send(\texttt{reward});
    q.dequeue();

    }
    }
}
contract MyQContract {

    Queue q = QueueContract(0x1d11e5fbe221);

    function foo() {
        ...
        q.enqueue(addr1);
        q.enqueue(addr2);
        someAddr.call.value(...);
        address i = q.dequeue();
        // Assuming i == addr1
        i.send(reward);
        q.dequeue();
    }
}

Transaction

mqc.foo(): enq(addr1) enq(addr2) deq() = ?

someAddr(): Any manipulation with q
Accounts using *smart contracts* in a blockchain are like threads using *concurrent objects* in shared memory.
Accounts using *smart contracts* in a blockchain are like threads using *concurrent objects* in shared memory.

<table>
<thead>
<tr>
<th>contract state</th>
<th>object state</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/send</td>
<td>context switching</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>(Un)cooperative multitasking</td>
</tr>
</tbody>
</table>
Reentrancy and multitasking

```solidity
// Burn DAO Tokens
Transfer(msg.sender, 0, balances[msg.sender]);
withdrawRewardFor(msg.sender); // be nice, and get his rewards
totalSupply -= balances[msg.sender];
balances[msg.sender] = 0;
paidOut[msg.sender] = 0;
return true;
}
```
Reentrancy and multitasking

```solidity
1010  // Burn DAO Tokens
1011  Transfer(msg.sender, 0, balances[msg.sender]);
1012  withdrawRewardFor(msg.sender); // be nice, and get his rewards
1013  totalSupply -= balances[msg.sender];
1014  balances[msg.sender] = 0;
1015  paidOut[msg.sender] = 0;
1016  return true;
1017  }
```

不幸的是，DAO的内部状态仍然表明账户仍然有资金，因为其账户余额尚未在第1014行被零化。因此，恶意的`msg.sender`可以发起第二次撤资，这就是通过调用DAO合约，然后在第1012行再次发送支付。实际上，恶意的`msg.sender`可以发起第三次、第四次等撤资，所有这些都会产生支付。只有在被支付多次后的余额接近于零时，账户才会被零化。

之前对该漏洞的分析表明，问题是由于递归或意外的重入。从狭义上讲这是正确的，但从广义上讲，我们正在看到的是通常在许多情况下表现像并发系统的顺序代码。

### 3.1 Atomic updates in shared-memory concurrency

图3显示了一个典型的例子（使用Java 8风格的伪代码），展示了错误使用的并发对象。这个对象应该实现一个“原子”计数器，具有`get`和`set`方法。图3左侧显示了并发计数器的实现显然是线程安全的（即，没有数据竞争），这要归功于使用`synchronized`关键字。但是，问题在于右侧多线程客户端代码的使用。

具体来说，当两个线程并行运行时，`thread2`线程中的`incr()`方法可能会发生在例如线程1的`c.set(a + 1)`调用之后，这将使`assert`语句无效，使得整个程序非确定性地失败！

问题在于，`incr()`在其顶部的`Counter`实现没有提供期望的“原子性”保证，客户端代码是假设没有干扰`incr()`方法，因此计数器`c`将被递增1，而`a`和`b`在执行过程中将保持不变。
DAO:

withdrawRewardFor()

_inv balances[msg.sender] = 0

_recipient.call.value(...) :

Manipulation with DAO

Inv(contract.state, balance)

Inv c.atomicMethod()

Inv Environment

Inv c.atomicMethod()

Inv Environment

Inv c.atomicMethod()
Accounts using *smart contracts* in a blockchain are like threads using *concurrent objects* in shared memory.

<table>
<thead>
<tr>
<th>contract state</th>
<th>object state</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/send</td>
<td>context switching</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>(Un)cooperative multitasking</td>
</tr>
<tr>
<td>Invariants</td>
<td>Atomicity</td>
</tr>
</tbody>
</table>
Querying an Oracle

Transaction 1:
- `c.prepareRequest()`
- `o.raiseEvent()`

Transaction 2:
- `o.respond()`
- `c.__callback(data)`
Querying an Oracle

Block N

Transaction 1

- `c.prepareRequest()`
- `o.raiseEvent()`

Block N+M

Transaction 2

- `o.respond()`
- `c.__callback(data)`
function enter() {
    if (msg.value < 50 finney) {
        msg.sender.send(msg.value);
        return;
    }
    warrior = msg.sender;
    warriorGold = msg.value;
    warriorBlock = block.number;
    bytes32 myid =
        oraclize_query(0, "WolframAlpha", "random number between 1 and 9");
}

function __callback(bytes32 myid, string result) {
    if (msg.sender != oraclize_cbAddress()) throw;
    randomNumber = uint(bytes(result)[0]) - 48;
    process_payment();
}
Accounts using **smart contracts** in a blockchain are like threads using **concurrent objects** in shared memory.

<table>
<thead>
<tr>
<th>contract state</th>
<th>object state</th>
</tr>
</thead>
<tbody>
<tr>
<td>call/send</td>
<td>context switching</td>
</tr>
<tr>
<td>Reentrancy</td>
<td>(Un)cooperative multitasking</td>
</tr>
<tr>
<td>Invariants</td>
<td>Atomicity</td>
</tr>
<tr>
<td>Non-determinism</td>
<td>data races</td>
</tr>
</tbody>
</table>
Reasoning about High-level Behavior of Contracts
(as of Concurrent Objects)
Temporal Properties

\[
Q \text{ since } P \overset{\text{def}}{=} \forall s s', s \rightarrow_{c^*} s', P(s) \Rightarrow Q(s, s')
\]

- “Token price only goes up”;
- “No payments accepted after the quorum is reached”;
- “No changes can be made after locking”;
- “Consensus results are irrevocable”;
- \textit{etc.}
• A Coq-based DSL for formally defining high-level contract behavior as of a “concurrent object”;
• Definitions of generic semantic contract properties;
• *Formal proofs* for several case studies (in Coq);
• Reasoning about contract/object composition;
• A verified compiler from the DSL to EVM;
• A compiler from Solidity to the DSL;
To take away

Accounts using **smart contracts** in a blockchain are like

*threads* using **concurrent objects** in shared memory.

- Understanding *intra-* and *inter-*transactional behavior;
- Detecting *atomicity violations* and *data races*;
- Repurposing *existing* verification ideas;

Thanks!