Solidity 0.5: when ***typed*** does not mean ***type-safe***

S. Crafa

Università di Padova, Italy

crafa@math.unipd.it

M. Di Pirro

Kynetics, Italy

matteo.dipirro@kynetics.com

FMBC - *11 October 2019 - Porto*

Agenda

- Smart contracts and Solidity
- Unsafe gambling game
- Safe gambling game
- Conclusion

Trusted Solidity contracts

- Smart contracts are intended to be automatically enforced
- Solidity
  - ○ **Statically typed language**
  - ○ Claimed to be “type safe”

- Solidity **programmers commonly use the compiler** to check type errors in the source code

Trusted Solidity contracts

➡ Unfortunately…

- **Solidity's type safety is limited**
- **`address payable`** is intended to prevent Ether transfers to smart contracts that are not supposed to receive money
  - **The compiler fails to enforce such semantics**!
- Incorrect contracts lead to gas losses and money indefinitely locked

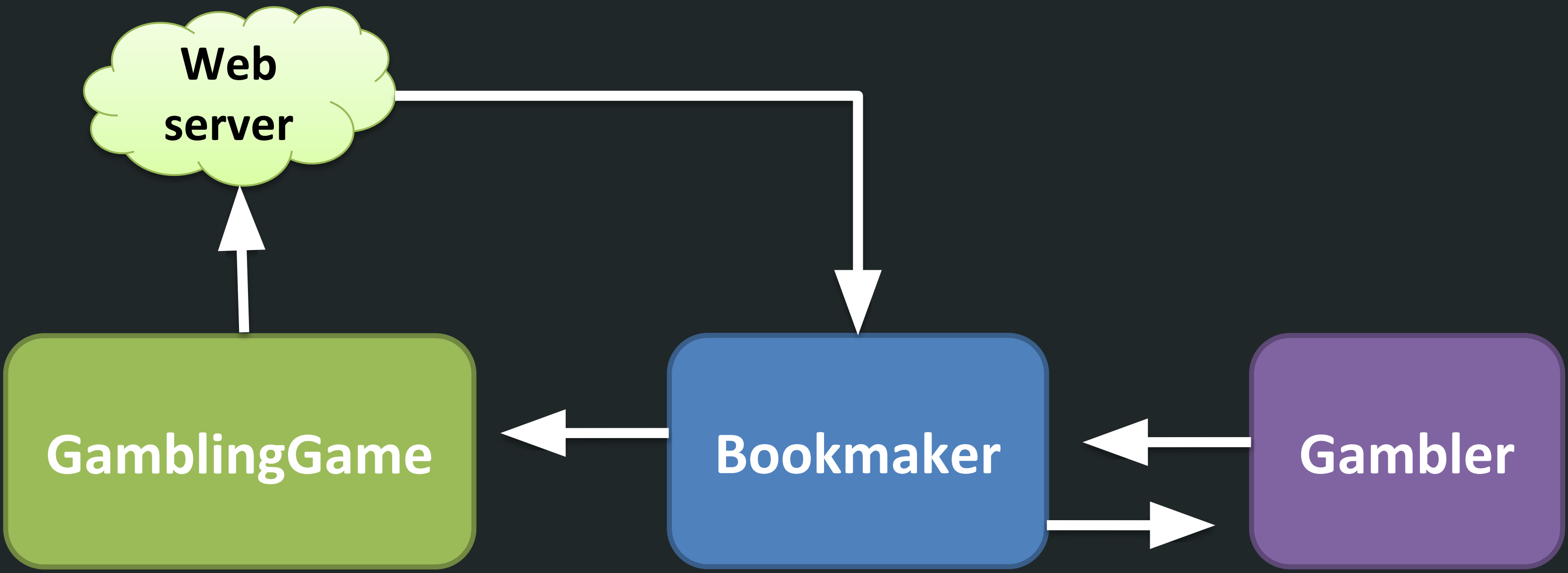
Trusted Solidity contracts

Unfortunately…

- **Solidity's type safety is limited**
- **`address payable`** is intended to prevent Ether transfers to smart contracts that are not supposed to receive money
  - **The compiler fails to enforce such semantics**!
- Incorrect contracts lead to gas losses and money indefinitely locked

Formal methods come to the rescue!

A gambling game



A gambling game

```
contract Gambler {
  constructor () payable public {}
  function bet(address bookmaker, string guess, uint n) external{
      require(amount < address(this).balance);
      Bookmaker(bookmaker).placeBet.value(n)(guess);
  }
}
```

A gambling game

```
contract Gambler {
  constructor () payable public {}
  function bet(address bookmaker, string guess, uint n) external{
      require(amount < address(this).balance);
      Bookmaker(bookmaker).placeBet.value(n)(guess);
  }
}
```

```
contract Bookmaker {
  mapping (address => uint) private currentBets;
  GamblingGame private game;
  constructor(address _game) public {game = GamblingGame(_game); }

  function placeBet(string guess) external payable {
    currentBets[msg.sender] += msg.value;
    game.play("http://...", guess, msg.sender);
  }
  function callback(...) external {...}
}
```

A gambling game

```
contract GamblingGame {
  event Play(address, string, string, address payable);
  function play(string url, string guess,
                address payable gambler) external {
      emit Play(msg.sender, url, guess, gambler);
      // eventually calls msg.sender.callback(outcome, gambler)
  }
}
```

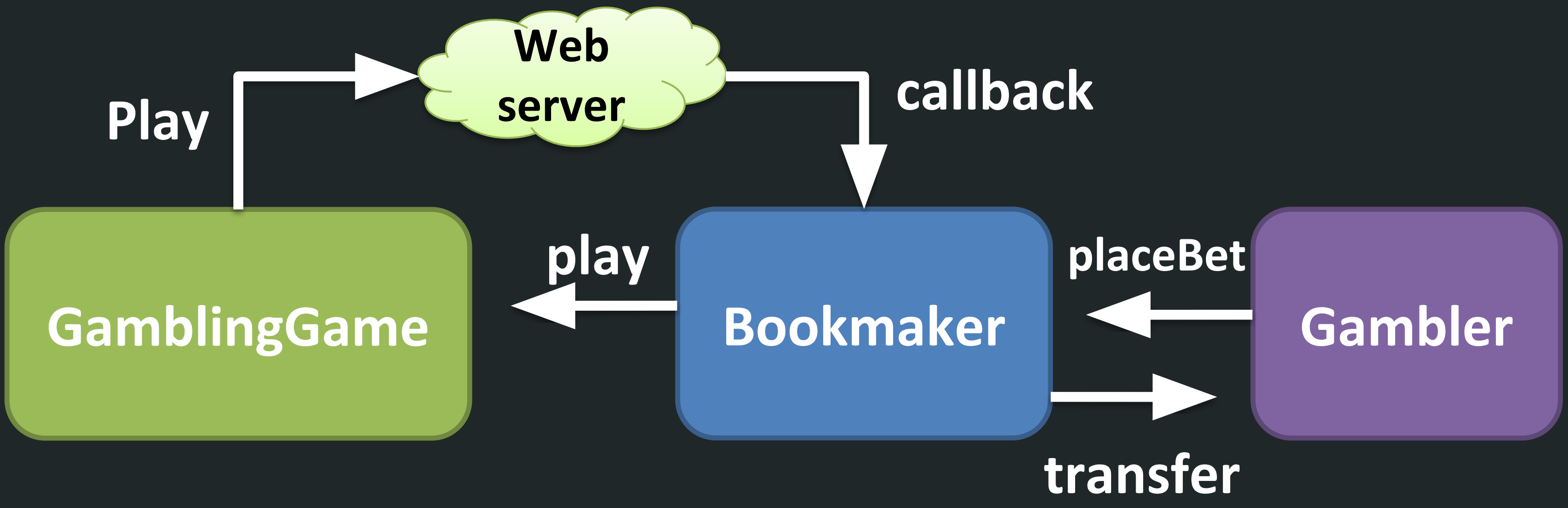
A gambling game

```
contract GamblingGame {
  event Play(address, string, string, address payable);
  function play(string url, string guess,
                address payable gambler) external {
      emit Play(msg.sender, url, guess, gambler);
      // eventually calls msg.sender.callback(outcome, gambler)
  }
}
```

```
contract Bookmaker {
  GamblingGame private game;
  function placeBet(..) external payable {...}

  function callback(bool outcome,address payable gambler) external{
    // if (outcome) gambler.transfer( ... )
    // otherwise gambler loses its bet
  }
}
```

A gambling game



- Gambler has **no fallback function**!
  - transfer will cause a runtime revert
  - Gambler’s bet indefinitely locked into Bookmaker

→ **Gambler’s code correctly compiles**

The compiler is happy

- **transfer** is defined on **address payable**

```
contract Bookmaker {
  function placeBet(string guess) external payable {
    ...
    game.play("...", guess, msg.sender);
  }
  function callback(bool outcome, address payable gambler) {
    // if (outcome) gambler.transfer( ... )
    // otherwise gambler loses its bet
  }
}
```

The compiler is happy

- `transfer` is defined on `address payable`

```
contract Bookmaker {
  function placeBet(string guess) external payable {
    ...
    game.play("...", guess, msg.sender);
  }
  function callback(bool outcome, address payable gambler) {
    // if (outcome) gambler.transfer( ... )
    // otherwise gambler loses its bet
  }
}
```

- `gambler` has type `address payable`!!

```
contract GamblingGame {
  function play(string url, string guess,
                address payable gambler) external {
    emit Play(msg.sender, url, guess, gambler);
  }
}
```

The compiler is happy

- **`msg.sender`** has **<u>always</u>** type **`address payable`**

  → But it will be substituted with a non-payable address

  → The use of `address` (`payable`) is unsound

  - ◆ Message-not-understood errors at run-time

The compiler is happy

- **`msg.sender`** has **always** type **`address payable`**
  - ➔ But it will be substituted with a non-payable address
  - ➔ The use of `address` (`payable`) is unsound
    - ◆ Message-not-understood errors at run-time

No Type Soundness!

Subject Reduction fails

Solidity 0.5 compiler is unsound

The problem…

- Solidity’s type `address` is an **untyped pointer**, like `void *`

- **Two features of Solidity** make this **problem pervasive**
  - Instances of smart contracts **can only be accessed through their public (“untyped”) address**
  - Extensive use of **msg.sender**
    - The **caller** is referred to through an **untyped** pointer
    - All the **callback expressions** undergo **potentially unsafe usages**

The problem…

- Solidity's type `address` is an **untyped pointer**, like `void *`
- **Two features of Solidity** make this **problem pervasive**
  - Instances of smart contracts **can only be accessed through their public ("untyped") address**
  - Extensive use of **msg.sender**
    - The **caller** is referred to through an **untyped** pointer
    - All the **callback expressions** undergo **potentially unsafe usages**

`msg.sender.transfer(n)` and `C(msg.sender).f()`

are **typical (dangerous!) Solidity patterns.**

…and the solution

1. **Refined address types**
   - `address<C>` is the address of contracts of type C *(or subtypes)*

2. **Refined function signatures** to **constrain** function **callers**
   - `function foo<C> (T x)` can be called only by contracts of type *(lower than)* C

3. This solution is retro-compatible with legacy Solidity code, allowing new, safer, contracts to interact with s.c. already deployed

…and the solution

1. **Refined address types**
   - ○ `address<C>` is the address of contracts of type C *(or subtypes)*

2. **Refined function signatures** to **constrain** function **callers**
   - ○ `function foo<C> (T x)` can be called only by contracts of type *(lower than)* C

Example:

Let `Top_fb` be the supertype of all the contracts providing a fallback

- `address<Top_fb>`
- `function foo<Top_fb>(T x)`

...and the solution

1. **Refined address types**
   - `address<C>` is the address of contracts of type C *(or subtypes)*

2. **Refined function signatures** to **constrain** function **callers**
   - `function foo<C> (T x)` can be called only by contracts of type *(lower than)* C

✓ Cast safety

✓ Transfer safety

Oracle pattern

```
contract GamblingGame {

  event Play(address<Bookmaker>, string,string, address payable);

  function play<Bookmaker>(string url, string guess,
                           address payable gambler) external {
      emit Play(msg.sender, url, guess, gambler);
      // eventually calls msg.sender.callback(...)
  }
}
```

`play` can be invoked only by a (subcontract of) **`Bookmaker`**

Oracle pattern

```
contract GamblingGame {

  event Play(address<Bookmaker>, string,string, address payable);

  function play<Bookmaker>(string url, string guess,
                           address payable gambler) external {
      emit Play(msg.sender, url, guess, gambler);
      // eventually calls msg.sender.callback(...)
  }
}
```

`msg.sender: address<Bookmaker>`

Transfer safety

```
contract Bookmaker {

  ...

  function placeBet(string guess) external payable payback {
    ...
    game.play(..., msg.sender);
  }
}
```

```
contract Gambler {
  ...

  function bet(...) external{
      Bookmaker(bookmaker).placeBet.value(n)(guess);
  }
}
```

Transfer safety

```
contract Bookmaker {

   ...

   function placeBet(string guess) external payable payback {
      ...
      game.play(..., msg.sender);
   }
}
```

```
contract Gambler {
   ...

   function bet(...) external{
         Bookmaker(bookmaker).placeBet.value(n)(guess);
   }
}
```

The call of `placeBet` in `Gambler` **does not compile**

Cast safety

```
contract Gambler {

  constructor () payable public {}

  function bet(address<Bookmaker> bookmaker,
               string guess, uint n) external{
      require(amount < address(this).balance);
      Bookmaker(bookmaker).placeBet.value(n)(guess);
  }
}
```

`bet` requires a `Bookmaker`

Cast safety

```
contract Gambler {

  constructor () payable public {}

  function bet(address<Bookmaker> bookmaker,
               string guess, uint n) external{
      require(amount < address(this).balance);
      Bookmaker(bookmaker).placeBet.value(n)(guess);
  }
}
```

The cast is safe

Conclusion

```
address        address payable        address<C>
```

In Solidity 0.5 **`address payable`** essentially provides only a refined **documentation** about addresses

- ○ The address of a contract that can “safely” receive Ether
- → Programmers expect that “safely” means “type-safely”

Conclusion

```
address        address payable        address<C>
```

In Solidity 0.5 **`address payable`** essentially provides only a refined **documentation** about addresses

○ The address of a contract that can “safely” receive Ether

→ Programmers expect that “safely” means “type-safely”

In [Crafa - Di Pirro - Zucca 19]
we **prove** the type soundness of this solution
on ***Featherweight Solidity***

Solidity 0.5
Typed does not mean **type-safe**

THANK YOU

Silvia Crafa
crafa@math.unipd.it

Matteo Di Pirro
matteo.dipirro@kynetics.com

Unsafe Gambling System

```
pragma solidity >= 0.5.0 <0.6.0;

contract Gambler {
  constructor () payable public {}
  function bet(address bookmaker, string calldata guess, uint amount) external {
      require(amount < address(this).balance, "Not enough balance for the bet");
      Bookmaker(bookmaker).placeBet.value(amount)(guess); }
}
contract GamblingGame {
    event Play(address, string, string, address payable);

    function play(string calldata url, string calldata guess, address payable gambler)
external {
        emit Play(msg.sender, url, guess, gambler); }
}
contract Bookmaker {
   GamblingGame private game;
   mapping (address => uint) private currentBets;

   constructor(address _game) public payable {  game = GamblingGame(_game); }

   function placeBet(string calldata guess) external payable payback {
        currentBets[msg.sender] += msg.value;
        game.play("...", guess, msg.sender);
   }
   function callback(bool outcome, address payable gambler) external {
        uint toBePaid = currentBets[gambler];
        currentBets[gambler] = 0;
        if (outcome && toBePaid != 0) {
            gambler.transfer(toBePaid + (toBePaid * 20)/100);
        }
        // otherwise msg.value is added to Bookmaker's balance
    }
}
```

Safer Gambling System /1

```
pragma solidity >= 0.5.0 <0.6.0;

contract Gambler {
    constructor () payable public {}

    function bet(address<Bookmaker> bookmaker, string calldata guess, uint
amount) external {
        require(amount < address(this).balance, "Not enough balance for
this bet");
        Bookmaker(bookmaker).placeBet.value(amount)(guess);
    }
}
contract GamblingGame {
    event Play(address<Bookmaker>, string, string, address payable);

    function play<Bookmaker>(string calldata url, string calldata guess,
address payable gambler) external {
        emit Play(msg.sender, url, guess, gambler);
    }
}
```

Safer Gambling System /2

```
contract Bookmaker {
    GamblingGame private game;
    mapping (address => uint) private currentBets;

    constructor(address<GamblingGame> _game) public payable {
        game = GamblingGame(_game);
    }

    function placeBet(string calldata guess) external payable payback {
        currentBets[msg.sender] += msg.value;
        game.play("...", guess, msg.sender);
    }

    function callback(bool outcome, address payable gambler) external {
        uint toBePaid = currentBets[gambler];
        currentBets[gambler] = 0;
        if (outcome && toBePaid != 0) {
            gambler.transfer(toBePaid + (toBePaid * 20)/100);
        }
        // otherwise msg.value is added to Bookmaker's balance
    }
```