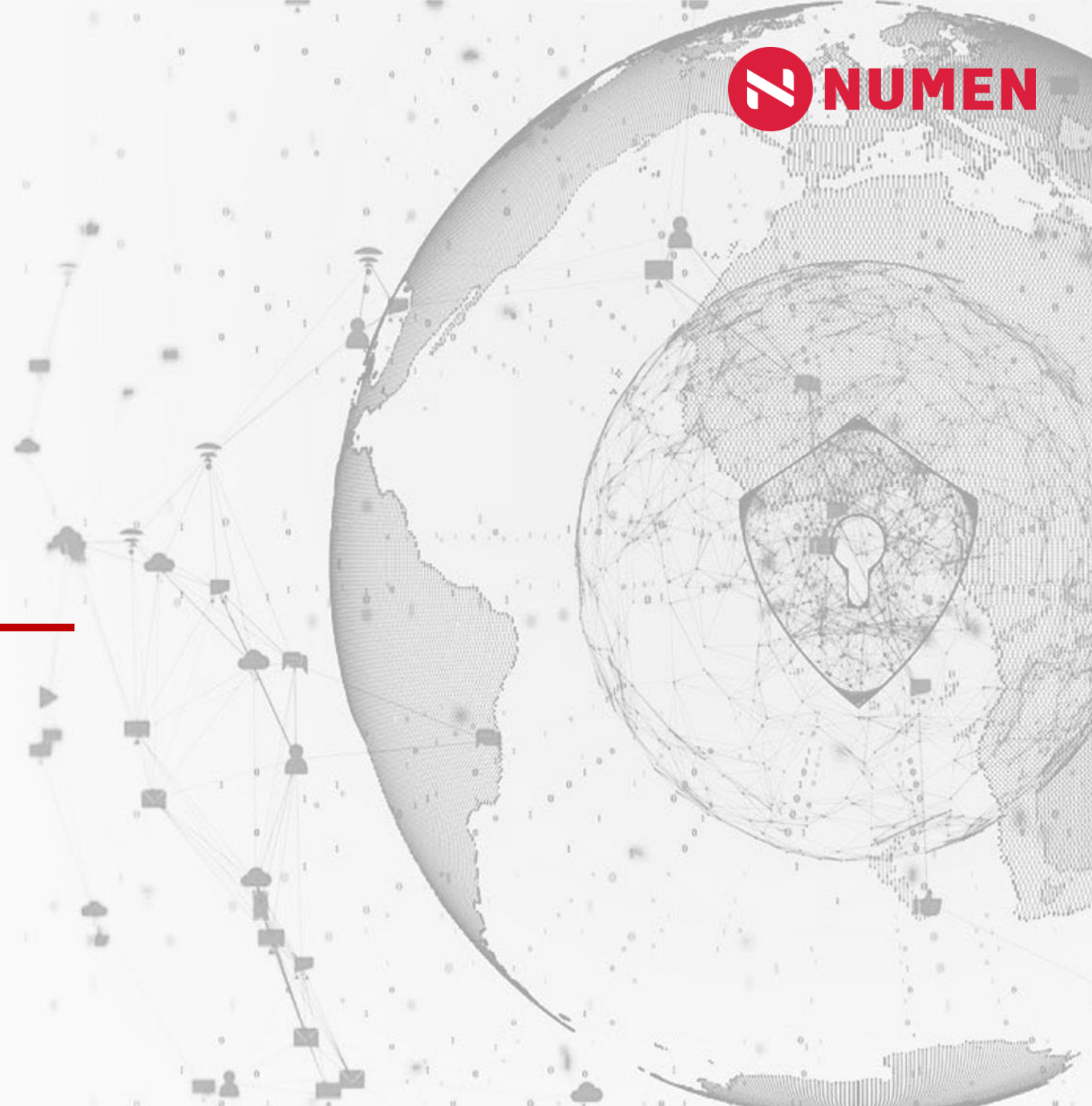


# Move Internals

---

Numen Cyber

Speaker: Nolan Wang



# 智能合约是非常规程序

## 智能合约实际上只做三件事:

- 定义新的资产类型
- 读取、写入和传输资产
- 检查访问控制策略

## 因此, 需要语言支持:

- 自定义资产、所有权、访问控制的安全抽象
  - 强隔离——编写安全的开源代码, 直接与有动机的攻击者编写的代码交互
- 传统语言中不常见的任务 :(
- 现有的 SC 语言不能很好地支持 :(

# 在其他智能合约语言中，通常不能

- 将资产作为参数传递给函数，或从函数返回一个
- 将资产存储在数据结构中
- 让被调用者函数临时借用资产
- 在合约 1 中声明一种资产类型，供合约 2 使用
- 在创建它的合约之外获取资产

\_\_\_永远“困”在其定义合约内的哈希map中

- 资产、所有权是智能合约的基本组成部分，但没有描述它们的词汇！
- Move 是第一个解决这个问题的智能合约语言(通过结构体，`move_to/borrow_global`，`abilities`等等)

# 通过子结构类型编码的资产和所有权

- “你给我一枚硬币，我就给你一个车名”

```
fun buy(c: Coin): CarTitle
```

- 如果你出示你的头衔并支付费用，我会给你一个汽车登记证

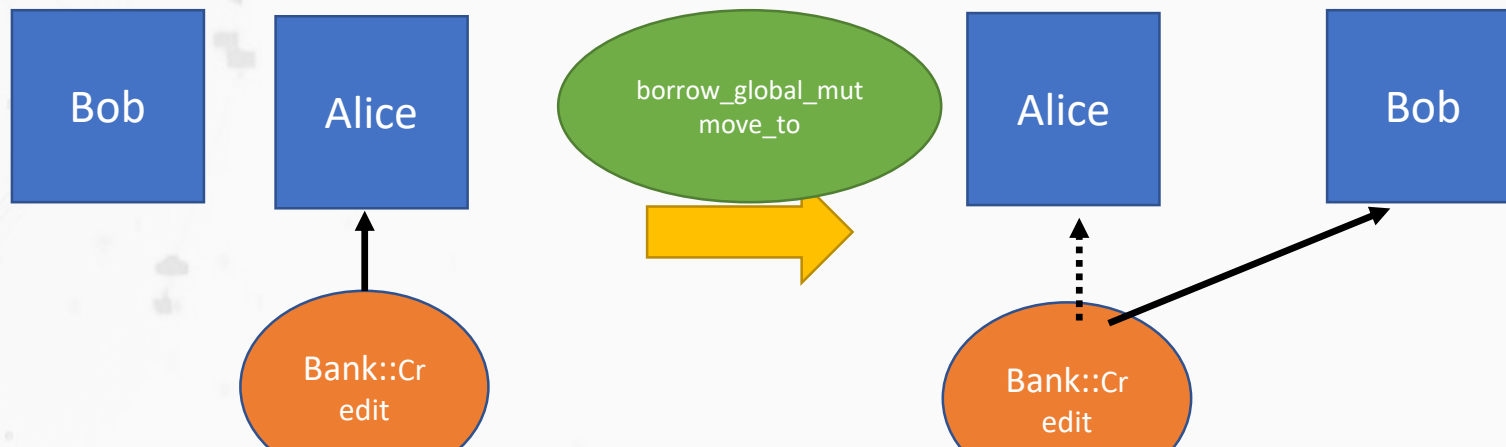
```
fun register(c: &CarTitle, fee: Coin): CarRegistration { ... }
```

CarTitle, CarRegistration, Coin 是用户自定义的，在不同模块中的结构体

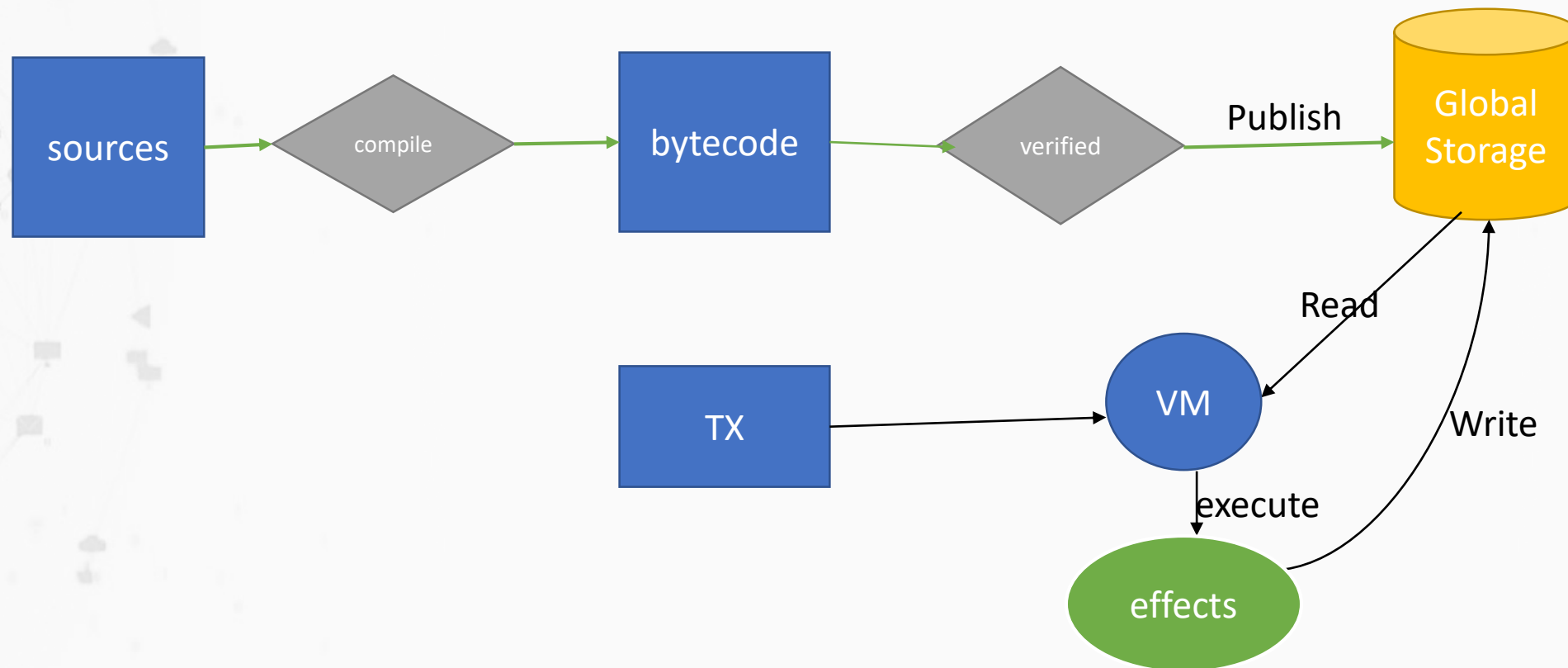
Move可以在不丢失完整性的情况下跨越信任边界

# 通过子结构类型编码的资产和所有权

- 表示货币可以使用用户定义的资源类型
- 资源即能力
- 资源使灵活的编程模式成为可能，而隐式的货币表示是不可能的。
- 线性类型系统



# Move生命周期

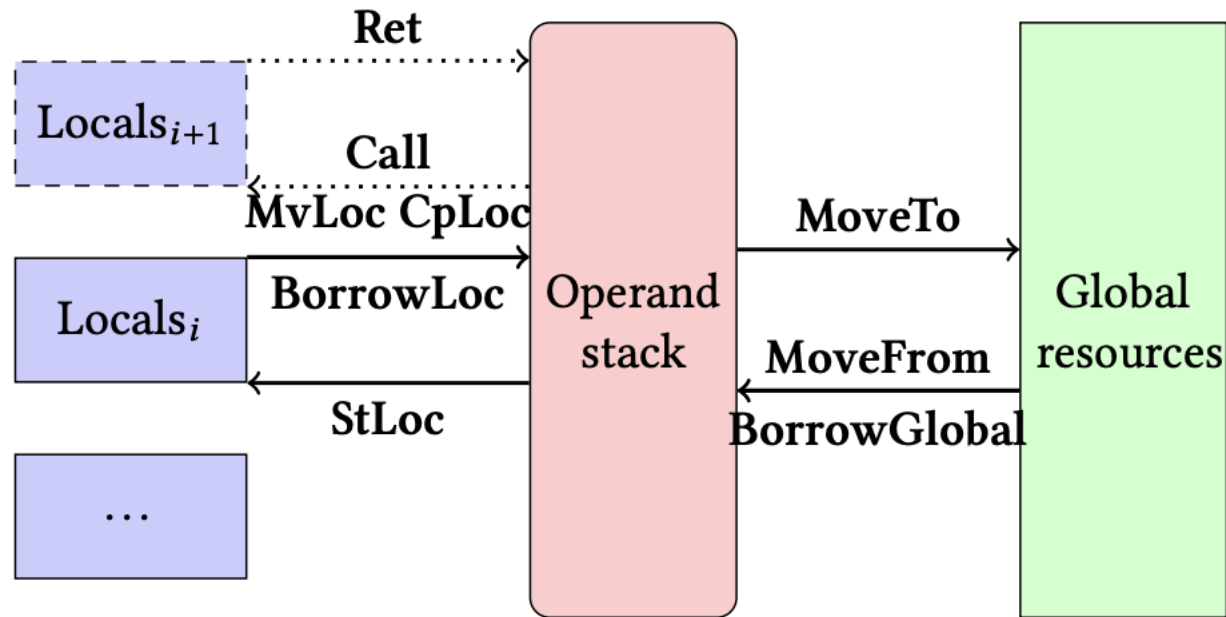


# Move源码结构

- ▶ move-analyzer
- ▶ move-binary-format
- ▶ move-borrow-graph
- ▶ move-bytecode-verifier ○
- ▶ move-command-line-common
- ▶ move-compiler
- ▶ move-core
- ▶ move-ir
- ▶ move-ir-compiler
- ▶ move-model
- ▶ move-prover
- ▶ move-stdlib
- ▶ move-symbol-pool
- ▶ move-vm ●
- ▶ testing-infra

```
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
  
Bytecode::BrTrue(offset: &u16) => {  
    gas_meter.charge_simple_instr(S::BrTrue?);  
    if interpreter.operand_stack.pop_as::<bool>()? {  
        self.pc = *offset;  
        break;  
    }  
}  
  
Bytecode::BrFalse(offset: &u16) => {  
    gas_meter.charge_simple_instr(S::BrFalse?);  
    if !interpreter.operand_stack.pop_as::<bool>()? {  
        self.pc = *offset;  
        break;  
    }  
}  
  
Bytecode::Branch(offset: &u16) => {  
    gas_meter.charge_simple_instr(S::Branch?);  
    self.pc = *offset;  
    break;  
}
```

# Move字节码执行



- 带非结构化控制流的堆栈虚拟机（使用goto等）
- 每个调用栈都有自己的局部变量
- 丰富的指令集，数据在操作栈上
- 指令的其他作用：
  - 在局部变量和操作数堆栈之间移动数据
  - 创建/销毁调用堆栈帧

local variable instructions	<b>MvLoc</b> $\langle x \rangle$   <b>CpLoc</b> $\langle c \rangle$   <b>StLoc</b> $\langle x \rangle$   <b>BorrowLoc</b> $\langle x \rangle$
reference instructions	<b>ReadRef</b>   <b>WriteRef</b>   <b>FreezeRef</b>
record instructions	<b>Pack</b>   <b>Unpack</b>   <b>BorrowField</b> $\langle f \rangle$
global state instructions	<b>MoveTo</b> $\langle s \rangle$   <b>MoveFrom</b> $\langle s \rangle$   <b>BorrowGlobal</b> $\langle s \rangle$   <b>Exists</b> $\langle s \rangle$
stack instructions	<b>Pop</b>   <b>LoadConst</b> $\langle a \rangle$   <b>Op</b>
procedure instructions	<b>Call</b> $\langle h \rangle$   <b>Ret</b>



# Move 字节码校验

- 类似JVM、CLR、proof-carrying代码的思路
- 类型安全
- 能力安全
  - E.g., 只有具有复制能力的类型才能使用 CopyLoc、ReadRef
- 引用安全
  - 无悬空引用, 无内存泄漏等
- 可简化的控制流图
- 本地变量安全性:
  - 所有访问的局部变量都已初始化+未移动
  - 栈平衡
  - 被调用者不能访问调用者的堆栈

# Move 字节码校验

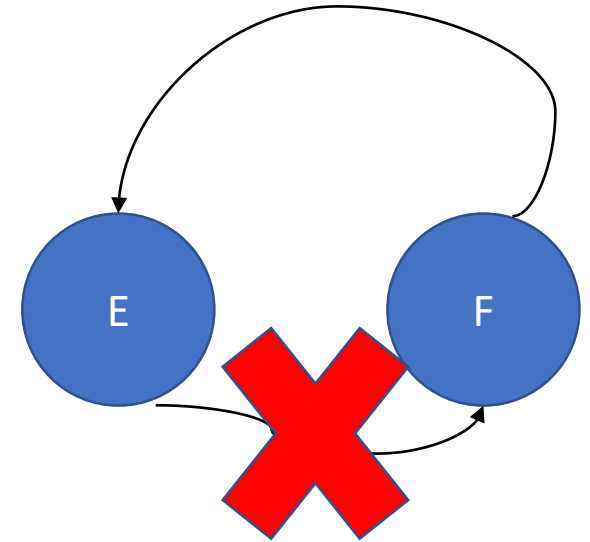
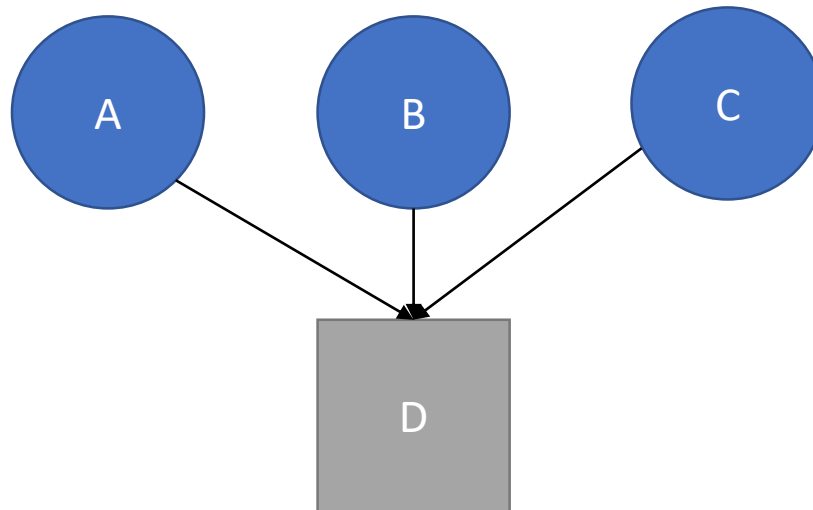
▼ move-bytecode-verifier	●
> bytecode-verifier-tests	
> fuzz	
> invalid-mutations	
▼ src	●
> locals_safety	
> reference_safety	●
> regression_tests	
⊗ ability_field_requirements.rs	
⊗ absint.rs	
⊗ acquires_list_verifier.rs	3
⊗ check_duplication.rs	
⊗ code_unit_verifier.rs	1
⊗ constants.rs	
⊗ control_flow_v5.rs	3
⊗ control_flow.rs	
⊗ cyclic_dependencies.rs	
⊗ dependencies.rs	3
⊗ friends.rs	

301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315

```
pub fn move_loc(  
    &mut self,  
    offset: CodeOffset,  
    local: LocalIndex,  
) -> PartialVMResult<AbstractValue> {  
    match safe_unwrap!(self.locals.remove(&local)) {  
        AbstractValue::Reference(id: RefID) => Ok(AbstractValue::Reference(id)),  
        AbstractValue::NonReference if self.is_local_borrowed(idx: local) => {  
            Err(self.error(status: StatusCode::MOVELOC_EXISTS_BORROW_ERROR, offset))  
        }  
        AbstractValue::NonReference => Ok(AbstractValue::NonReference),  
    }  
}
```

# Move linker

- 检查模块导入匹配声明
- 所有导入的函数都存在+具有预期的签名
- 所有导入的类型都存在+具有预期的能力
- 没有创建循环依赖



# MOVE验证器/VM 设计的关键

- 引用可以指向局部变量，但不能指向操作栈
  - 操作栈“拥有”它所用到的所有的值
- 没有持久引用——每个引用在单个 tx 中生成/销毁
- 所有值都是树形的：
  - 根是结构体或内置类型
  - 内部节点是结构体，字段是边，值是叶子
- 依赖关系图是一个DAG
- 没有fallback函数，没有重入，没有动态调用
  - 调用目标始终存在，并且是静态的

# Solidity vs Scilla vs Move

```

contract Bank
mapping (address => uint) credit;

function deposit() payable {
    amt =
        credit[msg.sender] + msg.value
    credit[msg.sender] = amt
}

function withdraw() {
    uint amt = credit[msg.sender];
    msg.sender.transfer(amt);
    credit[msg.sender] = 0;
}

```

```

contract Bank
field credit: Map Address Uint;

transition deposit()
    accept;
    match credit[_sender] with
        Some(amt) =>
            credit[_sender] :=
                amt + _amount
        None =>
            credit[_sender] := _amount
    end
end

transition withdraw()
    match credit[_sender] with
        Some(amt) =>
            msg = {
                _recipient: _sender;
                _amount: amt
            };
            credit[_sender] := 0;
            send msg
        None => ()
    end
end

```

```

module Bank
use 0x0::Coin;
resource T { balance: Coin::T }
resource Credit { amt: u64, bank: address }

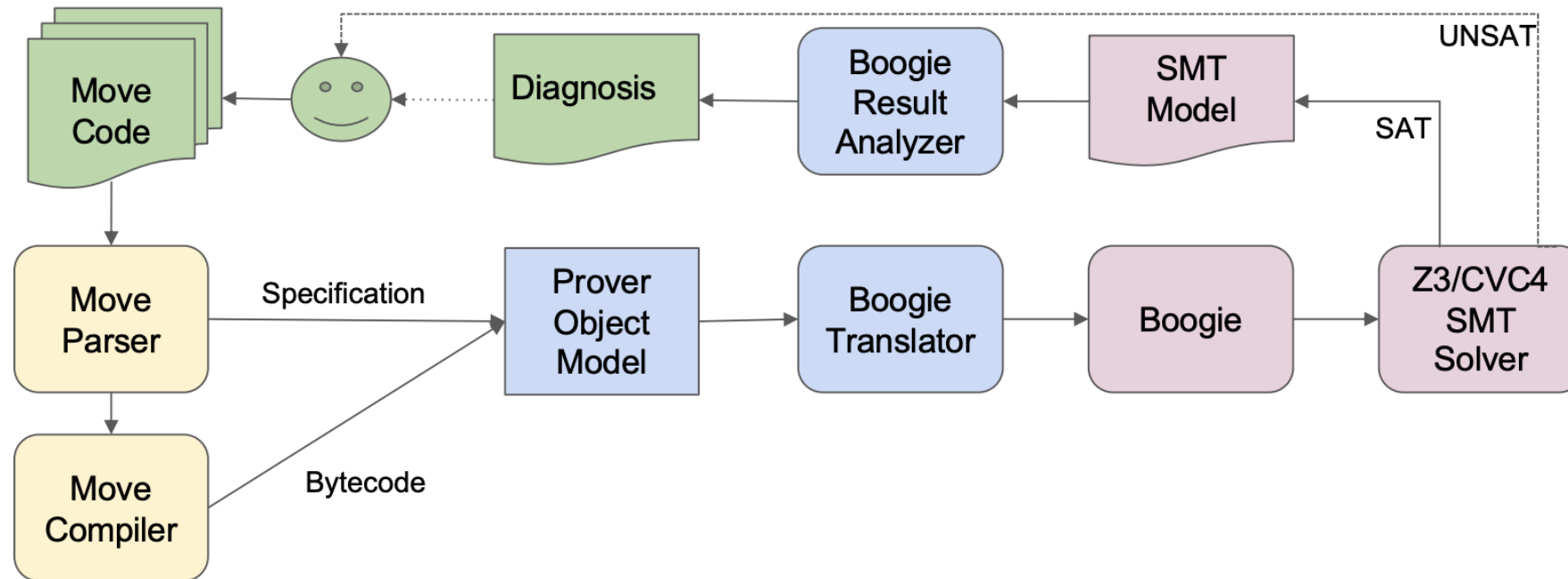
fun deposit(
    coin: Coin::T,
    bank: address
): Credit {
    let amt = Coin::value(&coin);
    let t = borrow_global<T>(copy bank);
    Coin::deposit(&mut t.balance, move coin);
    return Credit {
        amt: move amt, bank: move bank
    };
}

fun withdraw(credit: Credit): Coin::T {
    Credit { amt, bank } = move credit;
    let t = borrow_global<T>(move bank);
    return Coin::withdraw(
        &mut t.balance, move amt
    );
}

```

Fig. 1. A simple bank contract in Solidity (Left), Scilla (Middle), and Move (Right). Each code snippet represents

# Move Prover



**Fig.2.** The Move Prover architecture.

```
module M {  
  resource struct Counter {  
    value: u8,  
  }  
  
  public fun increment(a: address) acquires Counter {  
    let r = borrow_global_mut<Counter>(a);  
    r.value = r.value + 1;  
  }  
  
  spec increment {  
    aborts_if aborts_if !exists<Counter>(a);  
    ensures global<Counter>(a).value == old(global<Counter>(a)).value + 1;  
  }  
}
```

- [Move Specification Language \(MSL\)](#)

# Move Prover

# Move Prover

error: abort not covered by any of the `aborts\_if` clauses

```
└─ tutorial.move:6:3 ──
6 |   public fun increment(a: address) acquires Counter {
7 |       let r = borrow_global_mut<Counter>(a);
8 |       r.value = r.value + 1;
9 |   }
  |   ^
.
8 |       r.value = r.value + 1;
  |                               - abort happened here
=   at tutorial.move:6:3: increment (entry)
=   at tutorial.move:7:15: increment
=       a = 0x5,
=       r = &M.Counter{value = 255u8}
=   at tutorial.move:8:17: increment (ABORTED)
```



# Move 算术安全

## 使用Check\_\* 来阻止整数溢出

```
impl IntegerValue {
    pub fn add_checked(self, other: Self) -> PartialVMResult<Self> {
        use IntegerValue::*;
        let res: Option<IntegerValue> = match (self, other) {
            (U8(l: u8), U8(r: u8)) => u8::checked_add(self: l, rhs: r).map(IntegerValue::U8),
            (U16(l: u16), U16(r: u16)) => u16::checked_add(self: l, rhs: r).map(IntegerValue::U16),
            (U32(l: u32), U32(r: u32)) => u32::checked_add(self: l, rhs: r).map(IntegerValue::U32),
            (U64(l: u64), U64(r: u64)) => u64::checked_add(self: l, rhs: r).map(IntegerValue::U64),
            (U128(l: u128), U128(r: u128)) => u128::checked_add(self: l, rhs: r).map(IntegerValue::U128),
            (U256(l: U256), U256(r: U256)) => u256::checked_add(self: l, rhs: r).map(IntegerValue::U256),
            (l: IntegerValue, r: IntegerValue) => {
                let msg: String = format!("Cannot add {:?} and {:?}", l, r);
                return Err(PartialVMError::new(major_status: StatusCode::INTERNAL_TYPE_ERROR).with_message(msg));
            }
        };
        res.ok_or_else(err: || PartialVMError::new(major_status: StatusCode::ARITHMETIC_ERROR))
    }
}
```

# 类型系统阻止资产价值滥用

Protection against:

## Duplication

```
fun f(c: Coin) {  
  let x = copy c; // error  
  
  let y = &c;  
  let copied = *y; // error  
}
```

## “Double-spending”

```
fun h(c: Coin) {  
  pay(move c);  
  pay(move c); // error  
}
```

## Destruction

```
fun g(c: Coin) {  
  c = ... ; // error  
  return // error--must move c!  
}
```

通过线性类型系统来实现

## Move的安全性


- Move 是真正的game changer
- 安全性确实大幅提升（字节码验证, prover, 资源抽象, 线性类型系统等)
- Business logic bug 还会存在




**Thank you**  
for your attention

Contact us at

---

 +65 6355 5555

 [Contact@numencyber.com](mailto:Contact@numencyber.com)

Find us at

---

