Oracle-Supported Dynamic Exploit Generation for Smart Contracts

1. 摘要

ContraMaster通过改变交易序列，实现改变多个交易。ContraMaster使用数据流，控制流，和动态合约状态来指导其突变，然后监控目标合约程序的执行，根据通用语义测试预言机验证结果以发现漏洞；作为一种动态技术，它保证了发现的每个漏洞都违反了测试预言机，并且能通过生成攻击脚本来利用这个漏洞。对218个易受攻击的漏洞进行测试，没有显示任何误报。

Definition 1 合约和合约状态

`C := 〈c,bal,A,σ〉`， c ∈Addr是标识合约的唯一地址，bal ∈N是合约外部可见余额，A ∈ 2^{Addr}是一个集合参与者的账户地址，σ是内部合约状态，全局变量的类型一致评估值

Definition 2 交易

交易`t := 〈s,r,v〉`,如果成功执行， 发送账户余额s.bal中扣除金额v，将资金转移到地址r ∈Addr，将交易前后变量g分别表示为pre(g)，post(g)

Definition 3 平衡不变量

$$\langle c, bal, A, \sigma \rangle，\ \Sigma_{a \in A} m_\sigma(a) - bal = K$$

假设一个簿记变量，`m: Addr->N`，K为一个常数。余额不变性要求合约余额与所有参与者的簿记余额综合之间的差额在交易前后保持不变。入宫在定义中没有正确的更新簿记余额，则违反此不变量表面发生了不规则事件

Definition 4 交易不变量

$$\langle c, r, v \rangle, \Delta(m_\sigma(r)) + \Delta(r.bal) = 0$$

对于每笔转出交易，c是发送合约地址，$\Delta(x) = post(x) - pre(x)$,

交易不变量要求从合同的簿记余额中扣除的金额始终存入接收方的余额中，保证了交易两端的一致性。

簿记变量的识别

通常名称为balance，balanceOf。区别于其他变量的特征：（1）从账户地址到无符号整数的映射，`mapping（address=>uint*）`（2）在一个payable函数中至少更新一次（3）正常交易中从账户地址收到的金额应反映为该地址的余额增加

2. ORACLE-SUPPORTED FUZZING

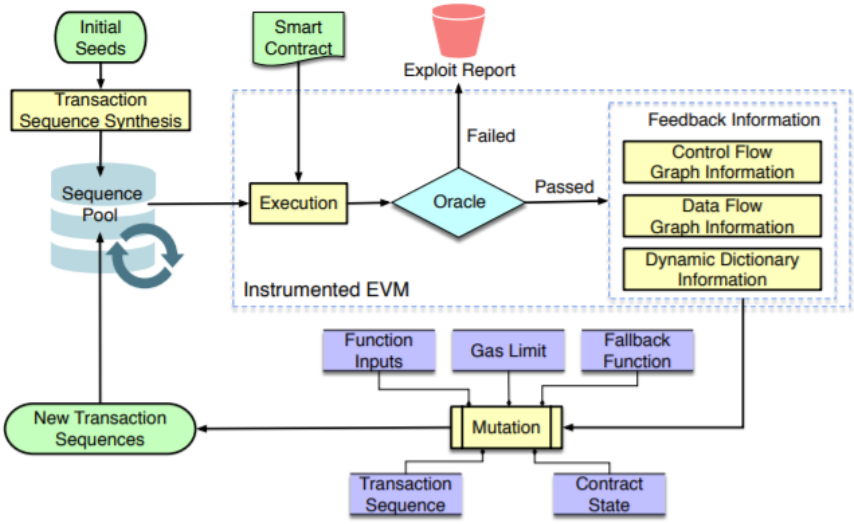

Fig. 6: Overview of the ContraMaster.

给定一组初始seeds，ContraMaster随机合成一组交易序列，并在模糊测试循环的每次迭代中从池中挑选一个。在已检测的EVM上按顺序运行每个交易。当交易完成时，ContraMaster根据语义测试预言机验证合约状态。如果检测到违规，ContraMaster会报告漏洞并提供生成的攻击合约和交易序列，可用于重现漏洞重现。否则CM会收集运行时执行信息，以指导下一次迭代的测试序列生成。收集的信息包括控制图、数据流图和合约状态信息。

（1）Oracle-Supported Fuzzing Algorithm

fuzzing组件的目标时自动生成违反测试预言机的交易序列。

Algorithm 1: Oracle-Supported Fuzzing

input : a contract program C, a set of initial seeds T
output: transaction sequences TS' violating the test oracle

```
1  TS ← generate initial transaction sequences from T ;
2  while time budget not reached and abort signal not received do
3      ts ← selectNext(TS) ;
4      E, dict ← ∅, ∅ ;
5      foreach transaction t_i in ts do
6          run t_i and collect execution trace e_i on EVM ;
7          dict ← extractValues(e_i) // dictionary values ;
8          E ← E, e_i (append e_i to E) ;
9          if test oracle is violated then
10             TS' ← TS' ∪ ts ;
11     foreach transaction t_i in ts do
12         if t_i has new branch coverage in e_i then
13             TS_i ← InputMutate(t_i, dict) ;
14             TS_g ← GasMutate(t_i) ;
15             TS ← TS ∪ TS_i ∪ TS_g ;
16             if t_i executes fallback function then
17                 TS_f ← FallbackMutate(t_i) ;
18                 TS ← TS ∪ TS_f ;
19     if E has new data dependence coverage then
20         TS_t ← TransSeqenceMutate(ts, E, dict) ;
21         TS ← TS ∪ TS_t ;
22     ContractStateMutate() ;
```

输入：一个合约C，一组初始种子T；输出：交易序列TS'，

由T生成一组初始交易序列TS，在模糊测试循环中，从TS中选择一个交易序列ts，将当前执行跟踪序列E和合约状态字典dict初始化为空。执行每个交易$t_i \in ts$，收集其执行轨迹e_i。将观察到的合约状态（如簿记变量，和合约余额的值）存到字典中。字典值稍后用于生成函数输入。每个交易e_i的执行轨迹被串联起来形成一个交易序列轨迹E。然后ContraMaster会检测是否违反预言机，若是，则将交易序列ts加

入到交易序列TS0中，这便是一个漏洞利用脚本（第10行）。

如果t_s到达了一个新的分支覆盖，我们在函数输入和燃料限制上执行变异算法，当fallback函数在交易中被调用同样在fallback函数上执行变异算法。

如果交易序列轨迹E有新的数据依赖覆盖，那就执行交易序列突变计算。

在最后通过合约状态突变计算，随机重置整个智能合约状态。

突出显示部分表现了ContraMaster与传统灰盒模糊测试的区别，传统模糊测试技术在单个调用上工作，ContraMaster可以在一个交易序列上工作。因为很多漏洞只能通过一序列的交易才能触发。并且CM对fallback函数进行了突变，攻击合约可以通过它与目标合约交互。

（2）突变策略

攻击合约：CM使用攻击合约来与目标合约交互。因此首先自动生成攻击合约。2-5行，使用变量来表示目标合约，在contructor中初始化。对于目标合约中每个函数，CM开发了一个代理函数来调用这个函数（5-11），最后合成12-14行的fallback函数。

```
1  contract attackDAO {
2    SimpleDAO public dao;
3    constructor(address _dao){
4      dao = SimpleDAO(_dao);
5    }
6    function donate(address to){
7      dao.donate.value(msg.value)(to);
8    }
9    function withdraw(uint amount){
10     dao.withdraw(amount);
11   }
12   ......
13   function () public payable {
14     dao.withdraw(...); //Reentrancy
15   }
16 }
```

Fig. 3: Reentrancy attack on the simple "DAO" contract.

函数输入：算法1的13行改变了传递给每个目标函数的参数。函数参数类型考虑两个：原始类型和数组类型。

原始类型包括布尔值 (bool)、帐户地址、无符号整数 (uint)、整数 *(int)* 和任意长度的原始字节数据 (byte*)。首先，从先前看到的，具有匹配类型的状态变量值的动态字典中，挑选特殊值来生成多个变异范围。在范围内，随机生成值作为候选函数输入；其次，机会性的否定这些输入中的bits以产生新的输入。对于数组类型，考虑固定长度和任意长度，使用与上述相同的技术为每个元素生成随机值。对于任意长度的数组，首先生成一个正随机数作为数组长度，然后再处理定长数组。对于任意长度的字节或字符串，使用字典中的值并通过位翻转来改变它们。

gas限制：在EVM上每条指令都有相关的费用gas。如果交易的gas超过了gas限制，就会抛出out-of-gas异常。为了模拟在交易中抛出异常的所有可能行为，在算法1的14行对gas限制进行了改变。首先估计目标交易的最大gas成本G_t和固有gas成本G_i，包括固定交易费用和数据相关费用。然后在G_t和G_i划分成n个相等间隔，并且随机选择一个gas限制来启动交易。

Fallback函数：这是以太坊的重要机制，与重入和异常混乱漏洞高度相关。当从被测目标合约收到资金时，攻击合约可能会使用特殊的回退函数来执行恶意活动。为了触发这些行为，CM在算法1的17行对fallback函数进行了修改。生成多个具有不同fallback函数的攻击合约来和目标合约交互，尤其是，允许目标合约的任何函数在攻击者的回退函数中被调用。除此之外，还有一个空的fallback函数和一个包含单个throw语句来触发异常混乱。

交易顺序：某些漏洞只能通过正确的交易序列触发，如DAO攻击只能通过先存入目标合约然后从其中退出来进行。为了找到成功的漏洞利用，改变调用序列如下。对于给定的候选交易序列：1，随机插入/从中删除交易；2，如果一个序列中的两个交易对同一个合约状态变量进行操作，将切换它们的顺序。

合约状态：交易的效果取决于发起交易的状态。为了改变合约状态，允许状态变量的值在多次测试中运行，并且定期重置状态，比如每n次交易后（因为某些合约状态对于发现新漏洞没有帮助）。合约状态的重置是通过将合约代码重新部署到私有测试网络来实现的，不会造成很大的开销。

（3）反馈机制

大致分为控制驱动，数据驱动，以及合约反馈状态信息。

数据驱动反馈：由于智能合约是与状态相关的，应该合成一个合适的交易序列来检测漏洞。但交易序列不能被控制流信息所引导，因为不同的交易序列不一定能覆盖新的CFG边。因此提出数据流来引导交易序列改变。如果改变的交易序列覆盖了新的数据依赖，它就是一个有趣的交易序列。首先定义数据依赖：

Definition 5 数据依赖 如果存在一个从*x*到*y*的有向路径*p*，那么就存在一个从*y*到*x*的数据依赖。*x*定义了一个变量$v \in V$, *y*使用*c*并且没有节点$z \in p$重新定义*v*。

如果一个序列中的两个交易对同一个合约状态变量进行操作，将切换它们的顺序：提款—>存款 交易序列都对簿记变量余额进行操作，因此可以切换它们的顺序以生成新的顺序“存款->提款”，这有可能会触发重入漏洞

合约状态反馈：在大多数情况下，当前交易的执行很大程度上取决于合约的状态，如“存款→取款”，重入漏洞的触发取决于存入金额是否大于提取金额。因此可以使用合约状态来指导函数输入的生成，使得提取金额少于存入金额，随后触发潜在漏洞。实际上CM将动态合约状态提取为动态字典，类似于VUzzer，但后者使用代码中的立即数作为静态字典。

（4）ContraMaster 示例

```
contract SimpleDAO {
  mapping (address => unit) public balances;
  function donate(address to) {
    balances[to] += msg.value;
  }
  function withdraw(uint amount) {
    require(balances[msg.sender] >= amount);
    msg.sender.call.value(amount)();  //Exception Disorder
    balances[msg.sender] -= amount;
  }
  function another_withdraw(uint amount) {
    require(balances[msg.sender] >= amount);
    msg.sender.send(amount)();  //Gasless Send
    balances[msg.sender] -= amount;
  }
}
```

Fig. 2: A simple contract susceptible to the “DAO” attack.

```
contract another_attackDAO {
  ......
  function () public payable {
    throw; //No reentrancy
  }
}
```

Fig. 4: Exception disorder example.

No.	Transaction Sequences	Attack Contracts	Feedback Mechanisms	Mutation Strategies
1	$ts_1 = withdraw(10) \rightarrow deposit.value(5)(*)$	*another_attackDAO*	data-driven	transaction sequence
2	$ts_2 = deposit.value(5)(*) \rightarrow withdraw(10)$	*another_attackDAO*	contract state	contract state
3	$ts_3 = deposit.value(5)(*) \rightarrow withdraw(3)$	*another_attackDAO*	control-driven	fallback function
4	$ts_3 = deposit.value(5)(*) \rightarrow withdraw(3)$	*attackDAO*	—	—

Fig. 7: Illustration of the discovery of the DAO attack by ContraMaster.

以图2为例，用图7所示的一组可能的生成交易序列来说明CM的工作流程。基于初始seeds，`{withdraw(10),deposit.value(5)(*)}`，CM随机生成一个交易序列`withdraw(10)→deposit.value(5)(*)`(NO.1)，假设攻击使用的是图4的another_attackDAO。在执行ts_1之后，通过分析执行轨迹的数据流来确定提款和存款函数之间对状态变量余额的数据依赖性。基于数据驱动反馈，在ts_1上采用交易序列改变策略来生成一个新的
$ts_2 = deposit.value(5)(*) \rightarrow withdraw(10)$

执行ts_2没有暴露可重入问题，因为提取金额大于存入金额，导致运行失败。CM检查合约状态并添加“（balances，5）”（存款金额）到动态字典中。CM使用合约状态作为反馈，生成三组提现输入参数：大于等于5，等于5，小于等于5。例如可以最终得到一个序列
$ts_3 = deposit.value(5)(*) \rightarrow withdraw(3)$，基于合约状态改变策略。

为了触发重入攻击，攻击合约需要一个合适的fallback函数，图4中的fallback没有重新进入withdraw函数。基于控制驱动反馈，CM对fallback进行修改，并生成一个新的fallback函数，如图三

```
1  contract attackDAO {
2    SimpleDAO public dao;
3    constructor(address _dao){
4      dao = SimpleDAO(_dao);
5    }
6    function donate(address to){
7      dao.donate.value(msg.value)(to);
8    }
9    function withdraw(uint amount){
10     dao.withdraw(amount);
11   }
12   ......
13   function () public payable {
14     dao.withdraw(...); //Reentrancy
15   }
16 }
```

Fig. 3: Reentrancy attack on the simple “DAO” contract.

至此已经成功找到一个攻击合约attackDAO，以及一个交易序列ts_4，当发生重入是，Ether将通过提款功能转移到攻击者账户，最后第九行的余额更新将下溢余额。由于簿记变量余额的值变得不一样，违反了余额不变量。

3.实施

使用以太坊客户端Aleth v1.8.0的C++实现来设置单节点私有区块链作为测试网络，使用Truffle Suite v4.1.14作为测试工具。

前端从一些种子开始，根据反馈机制执行突变以持续生成心中自。在后端，修改了Aleth EVM以监控运行时执行：通过在每个交易完成后asserting invariants（断言不变量）来强制执行测试预言机。

如果检测到不变违规，则将测试序列报告为漏洞利用，否则将执行数据流、控制流和合同状态分析以向前端提供反馈，前端继续生成新的测试输入。