一种兼容以太坊的区块链，可阐明并确保智能合约的设计层面安全属性

（工作草稿）

Nikolaj Bjørner、Shuo Chen、Yang Chen、Zhongxin Guo 微软研究院

Peng Liu、Nanqing Luo 宾夕法尼亚州立大学

摘要

智能合约是去中心化技术的关键组成部分，但安全漏洞和后门严重损害其可信度。为解决这一核心问题，我们提出一种技术，使程序员能够专注于设计层面属性，而非特定的低层攻击模式。我们提出的这项技术称为**定理携带交易**（TCT），结合了运行时检查和符号化证明的优势。在 *TCT* 协议下，每笔交易都必须携带一个定理，证明其符合所调用合约中的安全属性；区块链会在执行交易前检查该证明。*TCT* 的独特设计确保定理能够以高效的方式得到证明和检查。我们相信，*TCT* 有望在未来推动可证明安全的智能合约发展。因此，我们呼吁各方携手合作，共同实现这一愿景。

1. 引言

我们正站在一场去中心化革命的门槛上，这场革命有可能改变我们的社会。然而，首先必须克服一些重大障碍。其中一个障碍是，通过共识在所有区块链节点上执行已发布的智能合约，并不能提供足够的透明度和可信度。这是因为合约代码的行为通常过于复杂，难以加以约束。因此，智能合约中的非故意漏洞和蓄意设置的逻辑后门，已经导致了许多备受关注的事件，并造成了巨大的经济损失。遗憾的是，由于去中心化范式所遵循的“代码即法律”原则，交易不可篡改，损失也无法挽回。

为了增强智能合约的安全性，传统方法是依靠工具和/或人类专家，尽可能多地检测错误或已知漏洞，并加以修复，以防止合约产生非预期行为。在本文中，我们提出了一种不同的方法，使合约能够强制执行设计层面的安全属性，这些属性明确表达了合约的预期行为，并在运行时得到强制执行。为此，我们引入了一项名为定理携带交易（*TCT*）的技术。借助 TCT，每笔交易都必须携带一个定理，证明其遵守被调用合约所规定的安全属性。这确保了违反任何安全属性的交易都会被阻止，而不论已部署代码中是否存在任何隐藏的错误或未预期的漏洞。TCT 使智能合约代码能够明确声明设计层面的安全属性，并由区块链虚拟机（EVM 或类似的虚拟机）保证这些属性始终成立。TCT 的独特之处在于其设计：它结合了运行时检查和符号化证明的优势，确保定理能够以高效的方式得到证明和检查。一旦某个定理经过验证并被区块链接受，未来的交易就可以复用该定理，以极低的运行时开销确认自身的安全性。本工作草案旨在记录 TCT 的基本概念和技术思路。

1.1. 为什么设计层面属性是智能合约获得信任的关键？

从概念上讲，可信度意味着系统的行为符合其设计规范。然而，在处理现实世界中的安全问题时，主流方法往往将其简化为

枚举特定底层违规模式的问题。例如，许多安全问题被归类为“输入净化漏洞”，将根本原因归咎于程序员未能预见某种特定形式的恶意输入。这种事后补救式的做法自相矛盾，因为它要求程序员预料到每一种未曾预料的可能性。换言之，他们需要枚举非目标，而不是明确目标。以这种方式很难建立信任。

智能合约安全也面临同样的情况。例如，整数溢出和重入这两种彼此独立的底层模式，常被视为典型的安全漏洞。事实上，它们本身都不是安全漏洞。下面的代码来自 Uniswap 的实现，其中明确指出整数溢出是有意为之。至于“重入”，它实际上是“回调函数”这一术语的同义词——被调用模块中的函数在返回前调用（即重新进入）调用方模块。这种模式在许多系统中很常见且不可或缺，却突然以“重入”这个新名称被视为漏洞模式。

```
75 uint32 区块时间戳 = uint32(block.timestamp % 2**32); 76 uint32 已用时间 = 区块时间戳 – 上次区块时间戳；// 溢出是有意为之
```

（来源：https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol）

设计层面属性。设计层面属性将使程序员免于承担枚举这些底层漏洞模式所带来的负担和风险。在第3节中，我们将展示一个 ERC-20代币合约[3], ，它存在漏洞，可通过整数溢出和重入加以利用。让我们设想时间回到 2016 年，那时还没有人了解这些漏洞模式。程序员可以采取什么措施来保障合约安全？事实证明，即使不了解这些模式，只要程序员能够指定并强制执行表1中的设计层面属性，两种攻击都不会得逞。

表1：ERC-20代币合约的设计层面属性

$(\forall x{:}\text{地址} \mid 0 \leq \text{余额}[x] \leq \text{总供应量}) \wedge$ $(\sum_x(\text{余额}[x]) == \text{总供应量})$

这些属性声明，每个账户的余额都应为非负数，且所有余额之和应等于总供应量。从本质上说，它们只是阐明了程序员的设计目标。如果区块链能够强制保证这些属性，那么利用整数溢出和重入漏洞发起的攻击就会违反这些属性，从而被阻止。

这个时间旅行思想实验具有现实意义——可信度应当且能够在无需了解具体实现层漏洞模式的情况下建立；相反，关注重点应转向设计层面属性。

广泛多样的设计层面属性。信任问题不仅涉及易受攻击的合约，也包括那些企图欺诈用户的恶意合约。设计层面属性通过确保合约明确表达的保证得到遵守，来保护用户。在Uniswap早期历史中，正式阐明合约属性[5]是迈出的关键一步。可以合理地认为，这些明确的属性增强了用户的信心，促使Uniswap成为以太坊上领先的去中心化交易平台。

我们设想了许多类型的设计层面属性，它们可以保护合约的安全，避免非故意漏洞，并让用户确信合约的意图是良性的。然而，如今的区块链运行时难以对这些属性进行具体检查。TCT 支持在运行时对这些设计层面属性进行符号验证，使得强制执行这些属性切实可行且效率极高。

- 合约不变量——表1中的合约不变量等属性会在每笔交易前后得到保证。它们包含量词、映射求和等，因此难以进行具体检查。
- 精确的数学关系——例如，Uniswap 的设计从根本上依赖于一个线性流动性提供者关系（即 x 1 / y1 == x 2 / y 2 ），以及交易者的恒定乘积关系（即 x 1 * y2==x 2 * y2==k ）。这些设计层面属性无法进行具体检查，因为每笔交易都要收取少量交易手续费（例如 0.3%）。TCT 可以定义并检查在假设条件下的属性（例如，假设手续费比例为 0%），它会执行符号推理。同样，定点算术存在精度不足的问题，因此难以定义精确的数学关系。TCT 的符号推理不受此影响。
- *Modifies* 子句——程序员往往希望声明某个函数只打算修改一组变量，任何其他修改都不在预期之内。Dafny 语言引入的 Modifies 子句便是用于此目的的 [6]。曾发生过一起备受关注的智能合约漏洞，导致价值超过当时 $30+ 百万以太币被盗；漏洞源于 Parity 多重签名合约中的一条代码路径，该路径允许任意攻击者修改合约所有者。该事件已有分析 [8]。这条路径涉及"库合约"、"委托调用"和"默认函数"等底层细节，因此这类路径的存在令人意外。再设想一下，如果时光倒流，程序员事先指定了"默认函数"预期修改的变量，自然不会包括合约的所有者变量，那么即使不了解这些底层细节，也能阻止这次攻击。需要注意的是，Modifies 子句难以进行具体检查，因为这需要检查所有变量。再举一例，借助 Modifies 子句，NFT 代币合约可以向用户作出可验证的保证，证明它确实是灵魂绑定代币（即不可转让的代币） [7]。即使程序员试图在合约中隐藏一条用于转移所有权的秘密代码路径，任何执行该路径的交易也都必定会被拒绝。

- 其他被认为无条件成立的断言 – 对断言进行具体检查会产生运行时开销。程序员往往会"节省着"添加断言，并认为显而易见成立的属性属于"不必要"的断言。因此，大多数合约中只有零星的断言。借助 TCT，由于其符号证明机制，无条件成立的断言不会产生任何运行时开销。这将使断言与代码比率大幅提高，这对于智能合约的信任和安全性极具价值。例如，在 Uniswap 的代币交换交易中，一些中间账户需要暂时持有代币。交易结束时，这些账户的余额应为零。否则，任何用户都可以盗取剩余余额。全面的代码审计或许能确认这一属性对所有经过检查的代币对都成立，但添加一条 TCT 断言，就能以极高的可信度确认这一点，且不产生任何运行时开销。TCT 并不要求断言针对设计层面属性。我们希望程序员养成"尽量多加"的习惯，在设计和实现层面添加所有认为成立的断言。

1.2. TCT 为智能合约实现行为子类型

Liskov 和 Wing 提出的行为子类型 [9], 数十年来一直是面向对象编程（OOP）社区的追求。子类型要求如下。

> 设 $\phi(x)$ 是关于类型 T 的对象 x 可证明成立的性质。那么，对于类型为 S 的对象 y，若 S 是 T 的子类型，则 $\phi(y)$ 也应成立。

与支撑着当今智能合约及许多其他现实系统的传统面向对象编程子类型概念不同，行为子类型可确保派生类不会违反基类中定义的性质。继承意味着符合性，这种保证对于信任和安全性而言十分理想。例如，当前的 ERC-20 标准只规定了变量和函数，而函数行为则以英文描述，或隐含在函数名称中。借助行为子类型，ERC-20 标准可以被形式化地指定为一个包含不变量以及函数前置条件/后置条件的基类。假设我们定义一个名为 ERC20NonNegativeBalance 的基类，其中包含表1所列的性质，那么人们只需查看继承关系，就能轻松信任所有后代合约都是安全的。

然而，在如今的传统子类型机制中，继承并不意味着符合性。一个广为人知的例子是基类 Rectangle 及其派生类 Square [10]。对于基类而言，可以证明 Width-Height 的独立性，但这一已证明的属性不一定适用于派生类。有趣的是，代码“Rectangle r = Square(); r.setWidth(10); r.setHeight(5); assert(r.area()==50)”中的断言为假，这表明 Square 不是 Rectangle 的行为子类型/子类。众所周知，通过静态程序验证技术确保行为子类型十分困难，但对于 TCT 而言却很自然。

1.3. 我们的愿景

TCT 要求对现有的以太坊平台进行某些增强。我们计划构建一个名为 TCT-Eth 的兼容以太坊的区块链原型。它可以执行所有现有的智能合约，更重要的是，还能执行包含 TCT 风格断言（包括不变量）的新合约。我们需要朝两个目标努力：（1）吸引社区足够的关注，以成功运营 TCT-Eth；（2）向程序员和用户展示 TCT 技术的实用性，并增强他们对该技术的信心，从而使其能够应用于现实世界的部署。兼容以太坊还意味着，无论采用工作量证明、权益证明还是权威证明，去中心化的信任假设都保持不变。

我们展望智能合约未来将迎来一系列变革。

- “规范即代码之法”。区块链的现状是代码即法律。然而，代码的复杂性和缺乏透明度让人们付出了沉重代价。我们有必要通过形式化规范来约束代码的行为。如果合约的规范最终能够生效，那么用户同意合约就意味着仅同意该规范。当然，这是一种理想化的设想，但也是 TCT 致力于实现的方向。

- 程序员的关注重点。由于添加始终为真的断言不会产生任何运行时开销，程序员将通过专注于编写断言规范，大幅提高断言与代码比率。行为子类型将成为一项协作工作，不同抽象层级的程序员都能为之贡献力量。

- 让代码审计的商业接口更加清晰。代码审计是许多公司为智能合约开发者提供的一项重要服务。这项服务通常结合自动静态代码分析和人工代码检查。TCT 提供了一种新的商业模式，通过更清晰的接口连接审计公司与客户（即合约开发者）：后者提供代码和初始规范，前者则确定规范的最终版本，并

交付经过证明的定理，以确保所有预期交易都符合规范。我们相信，这种接口将帮助客户赢得终端用户更高的信任。

本文其余部分安排如下。第2节介绍启发了 TCT 的两项技术的背景知识。第3节通过一个易受攻击的代币合约，说明如何借助定理证明设计层面属性来阻止现实世界中的攻击。第 4 节介绍 TCT 协议。第5节将展示更具说服力的案例——Uniswap。第6节呼吁社区携手实现这一愿景。

2. 背景

TCT 的核心理念受到两项用于对现实世界系统进行形式化验证的技术启发——携带证明代码和具体符号测试。

携带证明代码（PCC）。PCC 旨在让操作系统内核能够安全地接收来自不可信方的代码模块 [11]。其创新之处在于，由不可信模块的提供者负责证明该模块符合操作系统内核声明的安全属性。受此启发，定理携带交易（TCT）是一种机制，要求每笔交易都携带一个定理，以证明其符合所调用的所有智能合约声明的安全属性。值得注意的是，与 PCC 不同，TCT 定理的粒度是每笔交易，而非每个代码模块。

具体符号测试。具体符号测试的核心思想是，对复杂程序的每条具体执行轨迹进行符号推理。例如，SAGE [13] 是一种模糊测试技术，已成功发现 Microsoft Windows 和 Office 代码中的大量错误。SAGE 被归类为白盒模糊测试技术，因为它会具体执行程序并记录执行轨迹，且针对它生成的每个输入都会如此。执行轨迹上所有分支条件的合取构成该轨迹的“路径条件”。为了生成下一个测试输入，模糊测试器会对部分分支条件取反，从而形成新的路径条件，并将其提交给约束求解器。这保证了每个测试输入都会产生一条新的代码执行轨迹，其效果远胜随机模糊测试方法。SAGE 的成功让我们相信，具体符号测试方法适用于智能合约执行，因为智能合约的代码复杂度远低于 Windows 和 Office 的 x86可执行文件。

3. 示例——存在安全漏洞的代币合约

我们的示例是一个包含两个漏洞的合约。第一个是整数溢出，与参考文献 [1] 中描述的情况类似。第二个是 [4] 中描述的重入漏洞。合约代码见附录 A，攻击代码见附录 B。图1展示了这些攻击。

3.1. 漏洞与攻击

在攻击1中，攻击者发起一笔调用 transferProxy 的交易。出乎意料的是，参数_value 和 _fee 分别被设为 2^255+1 和 2^255。在标记为“整数溢出”的代码行中，这两个参数相加，由于溢出，结果为 1，因此 require语句不会回滚。函数的其余部分也会继续执行。结果是，账户消息发送方和 _to 的余额变得极其庞大，导致这个代币生态系统出现严重的通货膨胀。

攻击2演示了这样一种后果：攻击者可以通过清除另一个余额为5的账户，使其某个账户的余额变为50。这是通过进行10次重入调用实现的，正如

图1所示。具体而言，当代币合约调用攻击者合约中的接收通知函数时，后者又调用前者中的清除函数。这一意外的代码执行顺序导致“balances[_to] += bal”被执行了10次。第一次重入攻击可追溯至2016年7月，是以太坊早期历史上的一个里程碑事件。攻击者窃取了360万个以太币。此外，这还严重损害了以太坊平台的信誉。这一事件的长期影响是，原有的“以太坊”社区此后分裂为“以太坊（ETH）”和“以太坊经典（ETC）”，[15], 这是一项沉重的代价。

如引言所述，整数溢出和重入只是运行时的两种底层细节，它们并不总是错误或漏洞。该代币合约之所以存在漏洞，是因为其中有一些意料之外的代码路径会导致违反表1中的属性。这些代码路径恰好利用了整数溢出和重入。

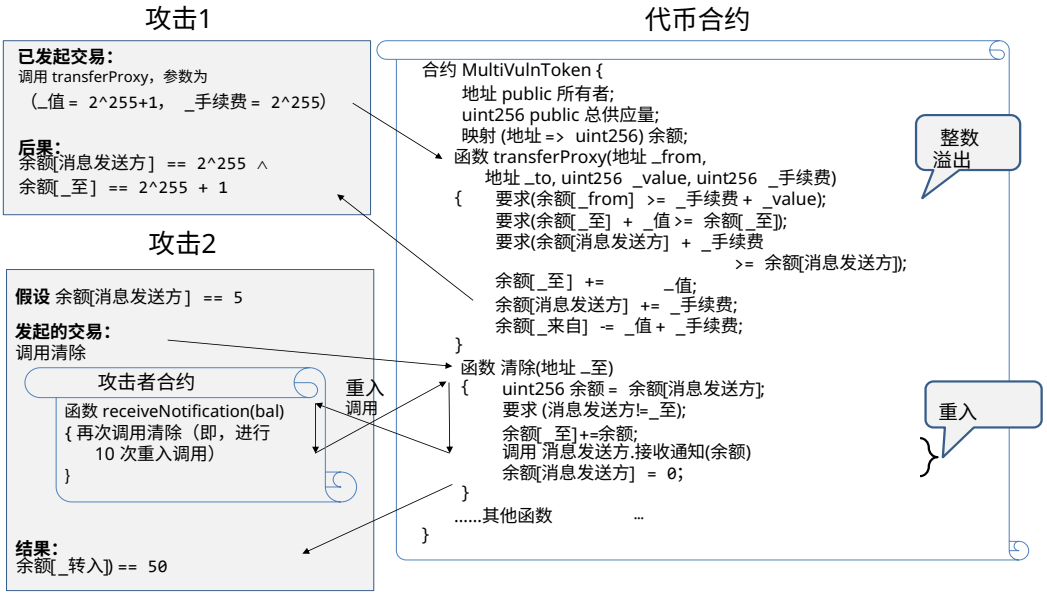


图1：利用存在漏洞的合约实施整数溢出和重入攻击。

3.2. 将代码路径的符合性证明为定理

现在，我们用形式化推理来考察这些攻击为何违反了表1中的属性。先来看攻击1。这笔交易的代码路径如下，是一段直线代码，我们用 Boogie 语言表示 [16]。

```
tmp1 := add(_手续费 , _value);
假设（余额[_来自] >= tmp1）；
tmp2：= add（余额[_to] ， _value）；
假设（tmp2 >= 余额[_to]）；
tmp3：= add（余额[消息发送方] ， _手续费）；
假设（tmp3 >= 余额[消息发送方]）；

余额[_to] ：= add（余额[_to] ， _value）；
余额[消息发送方] := add(余额[消息发送方] , _手续费);
余额[_from] := sub(余额[_from] , tmp1);
```

图 2：攻击1对应的直线代码路径

请注意，我们用公理来描述数学运算的基本属性，其中包括以下内容。这些公理并非针对我们研究的代码，而是数学中的普遍真理。

```
函数 sum(m: [地址] int256) returns (int256); //映射求和公理 (forall m: [地址] int256, a:地址, v:int256 :: sum(m[a:=v]) == sum(m)- m[a] + v); 公理 (forall m: [地址] int256 :: ((forall a:地址 :: 0<=m[a]) ==> (forall a:地址 :: m[a]<=sum(m)))); 函数 add(a,b:int256) returns (int256); //取模-2^256 加法 公理 (forall a,b: int256 :: add(a,b) == (a+b) % TwoE256); 函数 sub(a,b:int256) returns (int256); //取模-2^256 减法 公理 (forall a,b: int256 :: sub(a,b) == (a-b) % TwoE256);
```

图 3：基本数学运算的一些公理

表1 中的属性是不变量，必须在每笔交易前后都成立。因此，证明义务可表述如下：在代码路径开头添加两条“inv” assume 语句，并在末尾添加两条“inv” assert 语句。

```
（假设）假设（φ(v,s)）；（不变量）假设（sum(余额) == 总供应量）；（不变量）假设（forall x:地址 :: 0 <= 余额[x] && 余额[x] <= 总供应量）；tmp1 := add(_手续费 , _value); 假设（余额[_from] >= tmp1）；tmp2 := add(余额[_to] , _value); 假设（tmp2 >= 余额[_to]）；tmp3 := add(余额[消息发送方] , _手续费); 假设（tmp3 >= 余额[消息发送方]）；余额[_to] := add(余额[_to] , _value); 余额[消息发送方] := add(余额[消息发送方] , _手续费); 余额[_from] := sub(余额[_from] , tmp1); （不变量）断言（sum(余额) == 总供应量）；（不变量）断言（forall x:地址 :: 0 <= 余额[x] && 余额[x] <= 总供应量）；
```

图 4：直线代码路径的证明义务

“hyp”行指定了条件 φ(v,s)，在该条件下，不变量最终将成立；其中 v 表示交易的输入参数，s 表示区块链状态。

一个定理。假设合约程序员意识到，代码的意图是在条件 $0 \leq totalSupply < 2^{255} \wedge 0 \leq _value < 2^{255} \wedge 0 \leq _fee < 2^{255}$ 下处理交易。因此，“hyp”行指定为“assume (0<=_value && _value<TwoE255 && 0<=_fee &&_fee<TwoE255 && totalSupply< TwoE255)”。这段 Boogie 代码可以成功通过验证，这意味着以下定理（记为 τ）已得到证明。

定理 1：$\tau :=$ (MultiVulnToken ::transferProxy, $0 \leq totalSupply < 2^{255} \wedge 0 \leq _value < 2^{255} \wedge 0 \leq _fee < 2^{255}$, $0xa1b2c3d4$)

假设图 2 中的代码路径由安全哈希值 0xa1b2c4d4 唯一标识，那么该定理指出：对于以 MultiVulnToken::transferProxy 为入口函数的交易，如果满足条件 φ(v,s)，且代码路径的哈希为 0xa1b2c4d4，则该路径上的所有安全属性均成立。

值得强调的是，该定理对条件 $\neg \varphi(v,s)$ 并未作任何说明。在攻击1中，$_value == 2^{255}+1$，不满足 $\varphi(v,s)$。该定理并未表明攻击1会违反任何安全属性。从根本上说，TCT 并非漏洞发现技术，而是一种安全属性强制技术。第 4 节将说明 TCT 协议。它要求每笔交易都附带一个定理。由于攻击1凭空创造了余额，因此没有任何定理能够证明表1中的不变量，该交易也会因缺少有效定理而被 TCT 协议拒绝。

还值得强调定理的一般性。与具体符号测试类似，代码路径是具体的（即图 2 中由哈希 0xa1b2c3d4 标识的路径），但变量是符号化的。在第 4 节中，我们将说明 TCT 协议利用泛化来缓存并重用已证明的定理，这对于最大限度地降低成本十分重要。

攻击2。关于攻击2，也可以进行类似的分析。图 5 展示了直线代码。通常，标记有"(*)"的两行只应执行一次。然而，由于攻击者利用了重入模式，这段计算被重复执行了 10 次。与 2016 年以前的情况类似，程序员可能并不了解重入模式，但证明这条代码路径满足不变量是攻击者的责任。当然，这种尝试注定会失败。

```
      bal := 余额[消息发送方];
      假设(msg.sender != _to));
      tmp1 := add(余额[_to]，bal);
      余额[_to] := tmp1;
      /* 这些是由重入导致的 */
(*) tmp1 := add(余额[_to]，bal);
(*) 余额[_to] := tmp1;
      …
      tmp1：= add(balances[_to]，余额);
      余额[_至] := tmp1;
      /* 重入结束 */
      余额[消息发送方] := 0;
```

图 5：对应攻击2的直线代码路径

4. TCT协议工作流

在本节中，我们将介绍 TCT 机制的工作流。与当今的以太坊系统类似，其中有三方：交易发行方（例如浏览器钱包）、预执行服务（例如 *infura.io*）以及以太坊区块链。请注意，以太坊区块链由许多节点组成。预执行服务包含其中一个节点。

TCT机制会在每个以太坊节点中添加一个定理库，包括预执行服务的节点。定理的形式为 $\tau := (f, \varphi(v, s), \mathrm{h})$，，其中 f 表示一个合约函数；φ表示关于输入参数和区块链状态的布尔条件；而h是前文所述的路径哈希。该定理意味着，对于任何通过调用 f 发起的交易，只要满足 φ，若该交易顺利完成（即未被 EVM 回滚），且代码路径的哈希等于h，则可以保证执行过程中的所有断言均成立。从概念上讲，路径哈希等同于第 2 节提到的路径条件，因为二者都能唯一标识一条具体执行轨迹。不过，对于智能合约执行而言，路径哈希更易于表示和检查。

工作流 A。图 6 展示了交易发行方尝试向区块链提交一笔交易 Tx 时的工作流。与如今一样，Tx 首先会提交给预执行服务。该服务会尝试查找适用于 Tx 的定理。如果找不到这样的定理，预执行服务就会通知交易发行方：需要构造并证明一个定理，并且区块链预计会认为该定理适用于 Tx。如何实现这一目标由交易发行方自行决定。假设其定义了定理 τ 并构造了其证明 ρ。随后将元组 (Tx, τ, ρ) 提交到区块链。每个节点都会验证 ρ 是τ 的证明，并将τ 保存到其定理库中。然后，节点会检查τ 根据该节点的状态是否被视为适用于 Tx。若适用，则执行 Tx。

为了检查 ρ，节点首先需要通过测试运行交易 Tx 来获取 EVM 代码执行轨迹。随后，将记录的执行轨迹与哈希 h 进行比较，以确认是否匹配。获取代码执行轨迹后，节点会检查 ρ 是否能提供可靠推理，以证明“在条件 $\varphi(v,s)$ 下，代码执行轨迹不会违反任何断言”。

请注意，在此工作流中，交易发行方证明 τ（即生成 ρ）与区块链节点检查 ρ 是不同的机制。这类似于第2节中所述的携证明代码（PCC）概念。另一种方法是让双方使用相同的证明器。例如，可以使用 Boogie 作为证明器。交易发行方证明定理 τ，并在不提供 ρ 的情况下将其发送到区块链。区块链也使用 Boogie 重新证明并接受 τ。

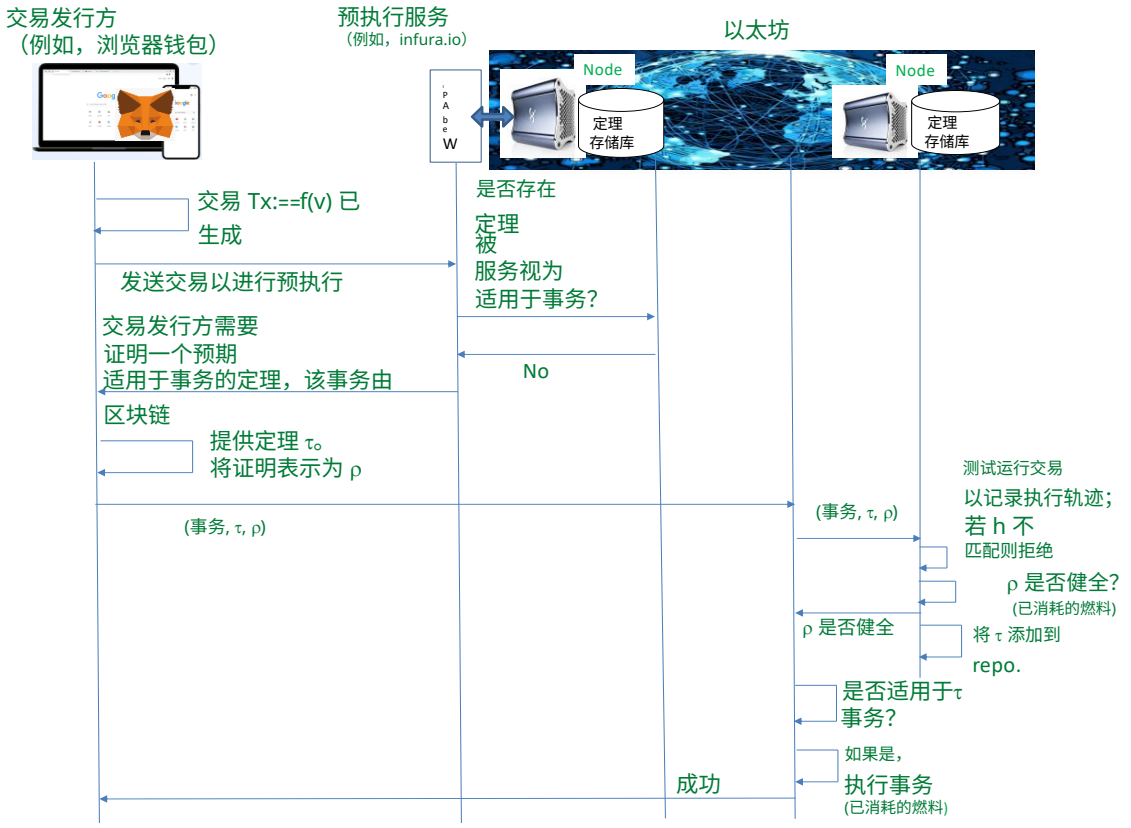


图 6：交易没有可用于证明其安全性的定理时的工作流。

工作流B。图 7 展示了另一种工作流。这一次，预执行服务找到了一个它认为适用于事务的定理 τ。元组 (Tx, τ) 被提交到区块链。每个节点只需

检查代码仓库中是否存在 τ 。如果存在，则执行前一个工作流中描述的其余步骤。

定理的一般性非常重要。假设第 3.2 节中的定理 1 先前已针对交易 MultiVulnToken::transferProxy(_from=...、_to=...、_value=10、_fee=1) 得到证明并被接受，而当前交易为 MultiVulnToken::transferProxy(_from=...、_to=...、_value=20、 _fee=2)。在这种情况下，TCT 将应用工作流B来复用该定理。唯一的运行时开销是从代码仓库中查找该定理，并检查条件“$0 \leq totalSupply < 2^{255} \wedge 0 \leq 20 < 2^{255} \wedge 0 \leq 2 < 2^{255}$”和“h == 0x a 1 b 2 c 3 d 4 ”。

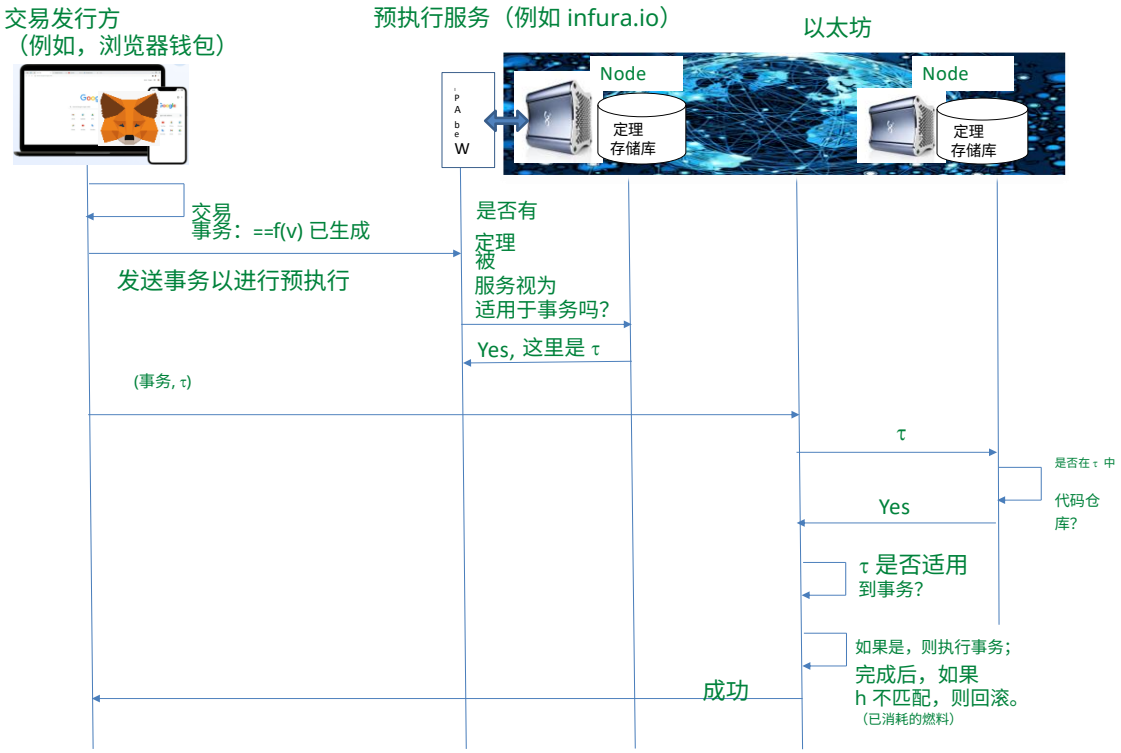


图 7：交易存在适用定理以证明其安全性时的工作流。

工作流 C。除上述两个工作流外，任何用户（通常是合约在测试阶段的程序员）都可以如图 8 所示，将任意定理 τ 及其证明 ρ 提交到区块链。如果 ρ是可靠的，τ 将被保存到代码仓库中。其唯一作用是将该定理填充到定理库中。该交易实际上不会被提交到区块链状态。

我们预计，合约的大多数定理都会通过工作流 C 被区块链接受，因为合约程序员应当运行覆盖其能够想象的所有正常场景的测试用例。合约开始处理真实用户的交易后，工作流B应当成为主要工作流，因为预计大多数交易都是正常的。只有当真实用户发起的交易经过所有测试用例中都未出现过的代码路径时，才会触发工作流 A。这可能是一次攻击。如果该交易是善意的，那么遗憾的是，用户需要承担提供新定理的负担。

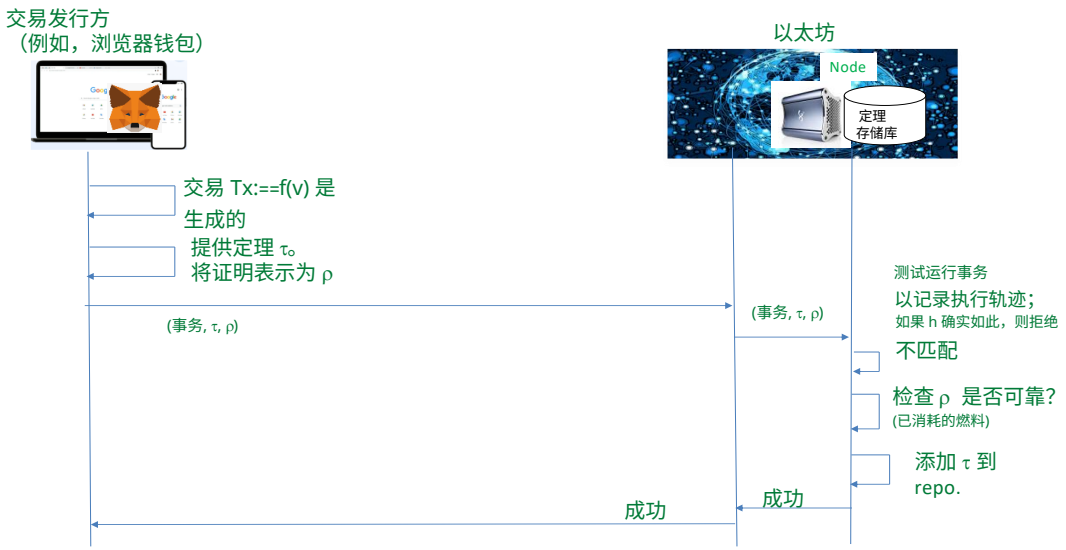


图 8：程序员提交定理及其证明的工作流。

5. 一个更有说服力的案例展示——Uniswap

第3节和第 4 节已经展示了使用 TCT 处理代币合约的可行性和价值，这令人振奋，因为代币是以太坊的一项主要应用。为了进一步展示 TCT，我们计划将其应用于 Uniswap 去中心化交易系统。根据已消耗的燃料数量 [18]，该系统目前是以太坊上使用最广泛的系统之一。Uniswap交易的复杂性明显高于单个代币交易，因为它通常涉及两种或三种代币。

近期事件暴露出中心化交易所缺乏透明度，使其能够随意操纵客户资金，例如 FTX 丑闻 [17]。因此，增强公众对去中心化交易所的信心至关重要。Uniswap 作为一个去中心化系统运行，其智能合约是公开可用的。如果能将设计层面的安全属性构建到这些合约中，就会有更多用户放心地使用该系统。

5.1. 协议简要说明

Uniswap 实现了一种自动做市商（AMM）协议，使持有不同 ERC20 代币的用户能够形成一个“市场”，并由该市场动态、自动地为这些代币定价。假设大多数用户都希望通过交易实现收益最大化，那么价格最终将达到代表公平市场价格的均衡。

Uniswap 已发展至 V3 版本。我们研究 V2，因为它已经包含了与交易所主要属性相关的大部分基本要素。Uniswap V2 协议的描述见参考文献 [20]。用户分为两类。第一类是流动性提供者（LP）。流动性提供者创建资金池作为储备，以促进不同 ERC-20 代币之间的交易。每个资金池都由一对代币定义。在我们的示例中，假设有一个由 AAA 代币和 BBB 代币组成的资金池。资金池的位置可以用坐标系表示，如图9所示。例如，(x1, y1) 表示该资金池当前持有 x1 个 AAA 代币和 y1 个 BBB 代币。除了创建资金池，流动性提供者还可以向已创建的资金池中添加或移除 AAA 代币和 BBB 代币。协议要求流动性提供者执行这些操作时，两种代币的比例保持不变。在图9中，这用虚线表示，虚线标注为“x1/y1=x2/y2”，经过 (0,0) 和 (x1,y1)，并延伸

通过 (0,0) 和 (x1,y1)。流动性提供者能够到达的新坐标记为 (x2,y2)，必须位于这条直线上（且仅限第一象限，不包括 (0,0)）。每个资金池都有自己的“流动性代币”。在任何时候，流动性提供者持有的流动性代币数量，都与她贡献的 AAA 代币占资金池中 AAA 代币总储备的比例成正比（这也等于她贡献的 BBB 代币占资金池中 BBB 代币总储备的比例）。

另一种用户类型是交易者。图9中的资金池支持交易者将 AAA 代币兑换为 BBB 代币，或反向兑换。假设当前坐标为 (x1,y1)。协议要求交易者沿着双曲线 x1*y1=k1 移动坐标，其中 k1 是一个常数。例如，(x1′,y1′) 是从 (x1,y1) 出发的有效目的地。移动到该目的地意味着交易者向资金池提供 Δx1=(x1′-x1) 个 AAA 代币，并从资金池获得 Δy1= (y1-y1′) 个 BBB 代币。[1]

当市场运行时，流动性提供者的操作与交易者的操作交错进行，因此其共同作用是使坐标在象限内移动，例如沿着路径 (x1,y1) -> (x2,y2) -> (x2′,y2′)。

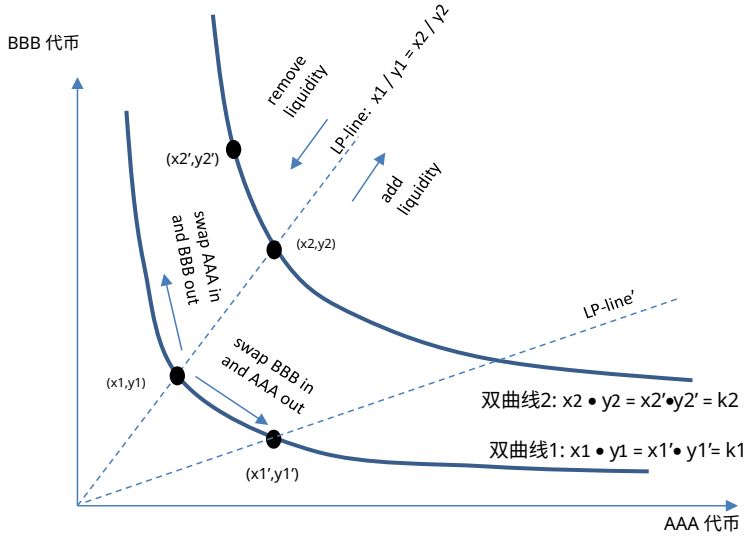


图9：坐标 (x,y) 在流动性操作和兑换操作后发生移动。

上述描述未考虑手续费。为了激励流动性提供者，每次兑换操作都会收取手续费，金额为兑换输入代币数量的 0.3%。例如，当交易者希望将坐标从 (x1,y1) 移动到 (x1′, y1′) 时，由于 Δx1 的 0.3% 会作为手续费被收取，因此 Δy1 会比上述不收取手续费的情形略小。详情见参考文献 [19] 第1节，但这与我们后续的描述无关。由于手续费会计入资金池的储备，双曲线会略微上移，因为 k1 略有增加。

5.2. Uniswap V2 的安全属性

我们希望 Uniswap V2 合约能够确保流动性提供者的投资“安全”，并确保交易者的交易“得到公平处理”。Uniswap 的场景涉及投资和交易，也就涉及市场。与 ERC-20 合约不同，我们不能要求“安全”和“公平”

[1] 该协议的经济学原理超出了本文描述的范围。基本而言，如果每位交易者都试图通过交易获利，坐标最终会稳定在一个均衡点，该均衡代表市场对 AAA 与 BBB 之间相对价格的共识。

意味着不会造成财务损失。关于合约应具备的属性，我们能提出哪些有意义的说法？直观来说，我们可以将一些属性描述如下。

- 每次 addLiquidity 操作都“按预期运行”。这意味着，在不收取手续费的情况下：(1) 坐标沿着流动性提供者线移动；(2) 如果某位流动性提供者调用 addLiquidity，存入 Δx 个 AAA 代币和 Δy 个 BBB 代币，并获得 Δl 个流动性代币作为奖励，那么只要没有发生交换交易，她随时都可以调用 removeLiquidity 并使用 Δl 个流动性代币，取回 Δx 个 AAA 代币和 Δy 个 BBB 代币。换句话说，许多流动性提供者可以自由地沿图9中的一条流动性提供者线移动坐标。只要没有交易者沿双曲线移动坐标，合约就能保证每个人都可以在想要的任何时候取回全部投资。（或许也可以这样表述这一保证：“任何流动性提供者的操作都不会影响其他流动性提供者”）

- 每次交换交易都“按预期运行”。这意味着，在不收取手续费的情况下：(1) 坐标沿着双曲线移动；(2) 如果某位交易者用 Δx 个 AAA 代币进行交易，并取回 Δy 个 BBB 代币，随后立即用 Δy 个 BBB 代币进行交易，那么她应该能取回 Δx 个 AAA 代币。
- 其他直观的属性还包括：“任何未参与的用户的余额都不会因任何流动性操作或兑换交易而改变”，以及“执行任何交易后，所有中间账户中的余额都必须为零（否则，任何用户都可能盗取这些余额）”等。

这些都是设计层面属性。正如我们在引言中所提示的，许多属性难以通过具体方式检查。例如，这些属性的定义假设不收取手续费，并且断言未来某笔交易发生后的假想状态（例如，未来执行 removeLiquidity 时，会取回当前 addLiquidity 中投入的代币）。若要通过具体方式检查这些属性，就必须实际执行假想的未来交易，这会改变区块链的具体状态。TCT 的符号证明特性使得定义和强制执行这些属性成为可能

这些属性。

6. 合作倡议：构建 TCT-Eth 区块链

我们的第一个里程碑是演示 TCT-Eth 区块链的测试网，以及支持 TCT 的浏览器钱包和预执行服务（如图 10 所示）。它们使用第 4 节中描述的 TCT 协议。在此测试网上，我们将演示使用经 TCT 增强的 Uniswap 合约交易 ERC-20 代币的去中心化交易端到端可行性。

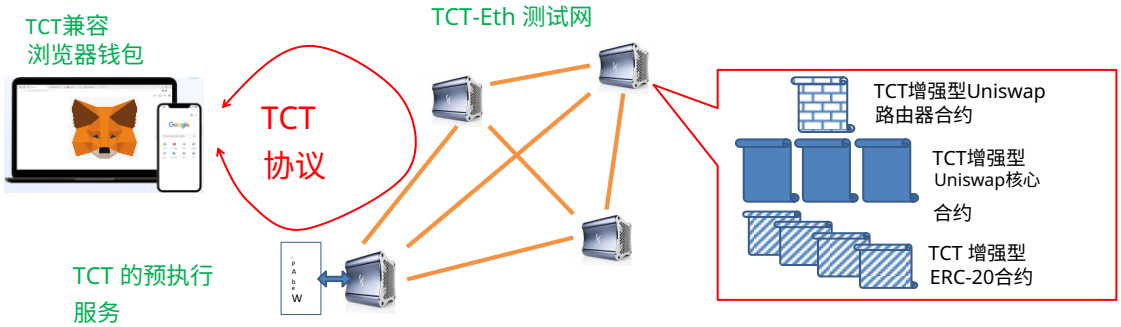


图 10：作为 TCT 首次展示项目的测试网和去中心化交易。

6.1. 主要组件和专业领域

为实现这一愿景，我们需要在以下专业领域寻求合作伙伴。

以太坊。我们需要增强 Go-Ethereum 实现 [26] ，使其支持 TCT。我们之前的项目 Forerunner [30] 已经实现了记录具体 EVM 代码路径并对其应用优化技术的功能。这是一个很好的代码库，TCT 项目可以在此基础上开展工作。我们需要记录代码路径，重新组装对应的Boogie直线代码，调用 Boogie 对其进行验证，并将其存储在定理库中。

Solidity 与 EVM。我们需要建立一种机制，让程序员能够指定 TCT 风格断言和不变量。我们的首要目标是设计一种适用于 Solidity 代码的方案。我们可以探索使用 Solidity元数据功能 [23], ，它允许通过 NatSpec 格式 [24] 的注释来标注每个合约、接口、函数等。程序员可以将合约不变量和函数后置条件实现为*TCT* 规范函数，并使用 @custom:TCT-invariant 和 @custom:TCT-post-condition 等自定义标签，将这些函数绑定到对应的合约和函数。

我们还需要具备 EVM代码分析方面的专业知识。这项技术可能类似于 Halmos EVM符号检查器 [25], ，但 TCT 只要求能够对直线代码进行符号执行。

形式化验证。在第一阶段，我们计划将 Boogie 用作验证预言机。本阶段的验证重点是研究图10中的智能合约。在第二阶段，我们将探索更深入的验证技术，包括线性/非线性程序的前置条件生成，以及证明对象生成（即生成可检查证明）。选择 Boogie 进行初步原型开发，是因为它已经存在，并且易于与基于文本的中间格式集成。

钱包和预执　　行服务。我们需要具备浏览器钱包和预执行服务方面的专业知识　　e.

6.2. 社区

鉴于项目的复杂性，建立稳固的合作关系至关重要。我们需要寻找这样的合作伙伴：(1) 真正拥有共同愿景和热情；(2) 具备能够作出贡献的专业知识；(3) 愿意为这项合作投入大量精力。幸运的是，这种情况在区块链/智能合约项目中很常见。例如以太坊基金会 [27], 、Uniswap 社区 [28] 、Zcash 基金会 [29], 等。

激励机制是建立成功合作关系的常见方式。这是我们目前缺乏经验的领域。除了共同愿景、对技术的热情以及推动科学进步的愿望之外，我们推测，围绕安全保证以及 TCT 所带来的信任提升而制定的商业目标，可以成为吸引利益相关者的基础。我们推测，围绕智能合约审计形式化方法而兴起的行业，可能会围绕预认证合约建立商业模式。

假设我们拥有一支实力雄厚的核心团队，就需要激励其他人成为贡献者，包括在 GitHub 上参与开源代码开发的系统程序员、编写智能合约的 Solidity程序员、为该项目提供资金的组织、运行区块链节点的组织，以及早期用户（类似 Uniswap 的早期用户）。我们需要精心设计激励合约及其相关的标准化或受支持格式，并将其部署在以太坊主网上。

参考资料：

[1] MESH 代币中的整数溢出漏洞 https://peckshield.medium.com/integer-overflow-i-e-proxyoverflow-bug-found-in-multiple-erc20-smart-contracts-14fecfba2759[2] Shur数。https://arxiv.org/abs/1711.08076

[3] 以太坊基金会。ERC-20 代币标准。https://ethereum.org/en/developers/docs/standards/tokens/erc-20/[4] 重入
The DAO 黑客攻击。https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/[5] Uniswap Labs。Uniswap 简
式化模型。https://blog.uniswap.org/uniswap- history#formalized-model[6] Jason Koenig 和 K. Rustan M. Leino。
入门指南。https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/krml220.pdf[7] E. Glen Wey
Puja Ohlhaver、Vitalik Buterin。去中心化社会：寻找 Web3 的灵魂。https://papers.ssrn.com/sol3/papers.cfm?abs
id=4105763[8] Lorenz Breidenbach、Phil Daian、Ari Juels 和 Emin Gün Sirer。深入剖析 Parity 多重签名漏洞。
https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/[9] Barbara Liskov 和 Jeannette Wing。“子类型
行为概念”。ACM 编程语言与系统汇刊。ACM 编程语言与系统汇刊，第 16 卷，第 6 期，1994 年。在线版本：
https://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf[10] Stefano Santilli。正方形与矩形之谜。
https://www.linkedin.com/pulse/squarerectangle- problem-stefano-santilli/?trk=pulse-article_more-articles_
related-content-card[11] George Necula 和 Peter Lee。无需运行时检查的安全内核扩展。USENIX 第 2 届操作系统设
现研讨会（OSDI）论文集，1996 年[12] 维基百科。具体符号测试。https://en.wikipedia.org/wiki/Concolic_testing[13
Patrice Godefroid、Michael Y. Levin 和 David Molnar。“自动化白盒模糊测试”。网络与分布式系统安全（NDSS）研讨
2008 年。[14] Morgan Peck。“即将进行硬分叉，将被黑客攻击的 DAO 资金返还给投资者”。IEEE Spectrum：科技，2
月19日。https://spectrum.ieee.org/hacked-blockchain-fund-the-dao-chooses-a-hard- fork-to-redistribute-funds[1
坊基金会。以太坊的历史。https://ethereum.org/en/history/[16] K. Rustan M. Leino。这就是 Boogie 2。
https://www.microsoft.com/en-us/research/wp- content/uploads/2016/12/krml178.pdf[17] 维基百科。FTX 破产。
https://en.wikipedia.org/wiki/Bankruptcy_of_FTX[18] Ultrasound Money - 销毁排行榜。https://ultrasound.money
Zhang、Xiaohong Chen、Daejun Park。常数乘积 (x × y = k) 做市商模型与实现的形式化规范。
https://github.com/runtimeverification/verified-smart- contracts/blob/master/uniswap/x-y-k.pdf[20] Uniswap 文
Uniswap V2 协议概览。https://docs.uniswap.org/protocol/V2/concepts/protocol-overview/how-uniswap-works[2
Hayden Adams、Noah Zinsmeister、Dan Robinson。Uniswap v2 核心。https://uniswap.org/whitepaper.pdf[22]
GitHub。Solidity 编译器。https://github.com/ethereum/solidity[23] Solidity 基金会。Solidity 合约元数据。
https://docs.soliditylang.org/en/v0.8.19/metadata.html[24] Solidity 基金会。NatSpec 格式。
https://docs.soliditylang.org/en/latest/natspec- format.html[25] Daejun Park。Halmos。
https://github.com/a16z/halmos[26] GitHub。Go Ethereum。https://github.com/ethereum/go-ethereum[27] 以太
会。“关于以太坊基金会。”https://ethereum.org/en/foundation/[28] Uniswap 社区。https://uniswap.org/commun
Zcash 基金会。https://zfnd.org/about/[30] Yang Chen、Zhongxin Guo、Runhuai Li、Shuo Chen、Lidong Zhou、
Zhou、Xian Zhang。“Forerunner：用于以太坊的基于约束的推测性事务执行。”ACM 操作系统原理研讨会，2021年10

附录 A：一个存在整数溢出和重入漏洞的代币合约示例 es

```
pragma solidity >=0.7.6; abstract contract Token { uint256 public 总供应量; function 余额查询(address _所有者) public view virtual returns (uint256 余额); } abstract contract 标准代币 is Token { function 余额查询(address _所有者) public view override returns (uint256 余额) { return 余额[_所有者]; } 映射 (address => uint256) 余额; } contract MultiVulnToken is 标准代币 { string public name = "演示代币，存在重入和整数溢出问题"; 实用工具库 _实用工具库; 构造函数 (uint256 initialSupply) pre post { 总供应量 = initialSupply; 余额[消息发送方] = 总供应量; 要求(success); } function transferProxy(address _from, address _to, uint256 _value, uint256 _fee) public returns (bool){ 要求(余额[_from] >= _fee + _value); 要求(余额[_to] + _value >= 余额[_to]); 要求(余额[消息发送方] + _fee >= 余额[消息发送方]); 余额[_to] += _value; 余额[消息发送方] += _fee; 余额[_from] -= _value + _fee; return true; } //此函数会将消息发送方的所有代币转入"_to"账户 function 清除(address _to) public { uint256 bal = 余额[消息发送方]; 要求 (消息发送方!=_to); 余额[_to] += bal; bool success; (success, ) = 消息发送方.call( abi.encodeWithSignature("接收通知(uint256)", bal) ); 要求(success, "通知消息发送方失败"); 余额[消息发送方] = 0; } }
```

附录 B：易受攻击的代币合约的利用

```
//合约重入攻击演示_attack { MultiVulnToken public multiVulnToken; address _to; uint count = 0; 构造函数 (MultiVulnToken _multiVulnToken, address __to) { multiVulnToken=_multiVulnToken; _to = __to; } function 接收通知(uint256) public { if (count < 9) { count++; multiVulnToken.clear(_to); } } function attack() public { multiVulnToken.clear(_to); } } contract 演示 { MultiVulnToken MultiVulnTokenContractAddress; address attacker1Address = address(0x92349Ef835BA7Ea6590C3947Db6A11EeE1a90cFd); 重入_attack attacker2Address1; address attacker2Address2 = address(0x0Ce8dAf9acbA5111C12B38420f848396eD71Cb3E); 构造函数() { MultiVulnTokenContractAddress = new MultiVulnToken(1000); attacker3Address1 = new 重入_attack(MultiVulnTokenContractAddress,attacker2Address2); //假设 attacker2Address1 初始拥有 5 个代币 MultiVulnTokenContractAddress.transferProxy(address(this), address(attacker2Address1),5,0); } function attack1_整数溢出() public { MultiVulnTokenContractAddress.transferProxy(address(this), attacker1Address, uint256(2**255+1), uint256(2**255) ); } function getBalanceOfAttacker1() view public returns (uint256){ return MultiVulnTokenContractAddress.余额查询(attacker1Address); } function getBalanceOfAttacker2() view public returns (uint256){ return MultiVulnTokenContractAddress.余额查询(address(attacker2Address1)) + MultiVulnTokenContractAddress.余额查询(attacker2Address2); } function attack2_重入() public { attacker2Address1.attack(); } }
```