ContractFuzzer：对智能合约进行模糊检测以进行漏洞检测*#

薄江†
计算机科学与工程学院
北京航空航天大学，
北京
jiangbo@buaa.edu.cn

刘烨
北京航空航天大学计算机科学与工程学院
中国北京
franklin@buaa.edu.cn

陈伟康
香港城市大学计算机科学系
wkchan@cityu.edu.hk

摘 要

分散式加密货币的特点是使用区块链在没有中央代理的情况下在网络上的同级之间转移价值。智能合约是在区块链共识协议之上运行的程序，使人们能够在达成协议的同时最大程度地减少信任。数百万个智能合约已部署在各种分散的应用程序中。这些智能合约中的安全漏洞对其应用程序构成了严重威胁。的确，以太坊平台上智能合约中的许多关键安全漏洞已给用户造成巨大的财务损失。在这项工作中，我们介绍了ContractFuzzer，这是一种新颖的模糊器，用于测试以太坊智能合约的安全漏洞。ContractFuzzer 基于智能合约的 ABI 规范生成模糊测试输入，定义测试 oracle 以检测安全漏洞，对 EVM 进行配置以记录智能合约运行时行为，并分析这些日志以报告安全漏洞。我们对 6991 个智能合约的模糊处理以高精度标记了 459 个以上的漏洞。尤其是，我们的模糊工具成功地检测到 DAO 合同的脆弱性，该脆弱性导致了 6000 万美元的损失，而 Parity Wallet 的脆弱性导致了 3000 万美元的损失和价值 1.5 亿美元的以太币冻结。

CCS 概念

• 安全和隐私→软件和应用程序安全；软件及其工程→软件测试和调试；

*该研究得到了国家自然科学基金委员会（项目号 61772056），中国工信部研究基金（项目号 MJ-Y-2012-07），研究资助理事会 GRF（项目号 11214116、11200015）的部分支持和 11201114），以及虚拟现实技术和系统国家重点实验室。

#ContractFuzzer 是开源的。https://github.com/gongbell/ContractFuzzer

†所有信件应寄给博江。

ASE '18，2018 年 9 月 3 日至 7 日，法国蒙彼利埃
©2018 年计算机协会。ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238177

关键字

Fuzzing，Fuzzer，智能合约，漏洞，Test oracle，以太坊，区块链

ACM 参考格式：

*姜波，刘烨和陈维坚。2018. ContractFuzzer：对智能合约进行模糊检测以进行漏洞检测。在 9 月召开的第 33 届 IEEE / ACM 国际自动化软件工程国际会议（ASE'18）上，3-7，蒙彼利埃 法国，10 页面。*https://doi.org/10.1145/3238147.3238177

1 介绍

近年来，加密货币和区块链技术在行业和学术界获得了极大的普及和关注。到 2017 年底，加密货币市场总资本已达到约 6000 亿[19]。加密货币通常采用经过精心设计的共识协议，该协议由其基础网络中的所有参与节点同意，并且网络的计算节点（即，矿工）负责在分布式交易中记录交易后的网络状态。共享分类账--一个区块链。最初提出区块链是为了在不信任的情况下在网络同伴之间进行价值转移[11]。后来，有许多增强的区块链平台支持智能合约。最受欢迎的之一是以太坊[22]，它以图灵完备的编程语言增强了区块链平台，允许开发人员编写智能合约和去中心化的应用程序。以太坊的生态系统正在快速增长：智能合约和去中心化应用程序的数量在 2018 年 3 月已增加到超过 200 万[28]。以太坊生态系统的爆炸性增长表明了其在孵化杀手区块链应用程序方面的潜力。

智能合约可以在区块链共识协议之上构建去中心化应用程序，以便用户可以在最小化信任的同时通过区块链达成协议。它们是在区块链上运行的代码，可以定义任意规则来控制数字资产[22]。分散式应用程序（DApp）基本上由一组智能合约作为后端，以及一组用户界面作为前端。智能合约已在实践中启用了广泛的 DApp，例如钱包，预测市场，即时消息，微博，众筹等。以太坊账户（包括智能合约账户和外部自己的账户）现在已

管理着 9,800 万以太币, 在 2018 年初约合 5,900 万美元[28]。

但是, 根据智能合约管理如此多的财富, 也使它们成为黑客攻击的诱人目标。确实, 智能合约的安全性问题已导致区块链社区遭受严重损失。臭名昭著的 DAO 合同错误[17]导致美国损失了 6000 万美元。奇偶钱包有两个漏洞[32] [33]。第一个损失了 6000 万美元, 第二个损失了超过 1.5 亿美元的以太币。

有几种原因使智能合约容易受到安全攻击。首先, 智能合约的每次执行都依赖于底层的区块链平台以及其他合作智能合约的执行。如果智能合约开发人员无法完全理解那些智能合约之间的隐式关系, 他们可能会轻松编写易受攻击的智能合约。其次, 编程语言 (例如, Solidity) 和运行时环境对于许多开发人员来说是新的, 并且这些工具仍然很粗糙。如果开发人员无法很好地处理工具链的缺陷, 漏洞可能会泄漏到智能合约中。最后但并非最不重要的一点是, 智能合约的抗扰性使其难以在部署后进行更新。尽管采取了一些变通方法[34], 但对智能合约的安全漏洞修复可能需要花费很长时间才能应用, 这会使易受攻击的智能合约面临安全威胁。

在先前的工作中, 提出了几种智能合约验证工具来检测智能合约中的安全漏洞。但是, 它们仍然有局限性。首先, 检测策略可能不够精确, 这可能导致较高的误报率, 即检测到的漏洞不太可能显现或无法利用。其次, 以符号方式验证所有可能的路径都会遭受路径爆炸问题, 如果仅验证某些路径, 这也可能导致假阴性。

在这项工作中, 我们提出了 ContractFuzzer, 这是一个用于检测以太坊平台上智能合约中的安全漏洞的模糊测试框架。ContractFuzzer 分析智能合约的 ABI 接口, 以生成符合测试中智能合约的调用语法的输入。我们针对不同类型的漏洞和工具 EVM 定义了新的测试预言机, 以监视智能合约的执行情况以检测实际的智能合约漏洞。然后, 我们还通过在我们的测试网上为 ContractFuzzer 部署 6991 个现实世界的智能合约来展示 ContractFuzzer 的实用性和适用性, 以对 ContractFuzzer 进行安全性测试。实验结果表明, ContractFuzzer 可以非常高精度地检测到 459 个以上的漏洞, 其中每个漏洞都已通过我们的人工分析得到了确认。

这项工作的主要贡献是三方面的。首先, 据我们所知, 这项工作提出了第一个模糊框架, 用于检测以太坊平台上智能合约的安全漏洞。其次, 它提出了一组新的测试 oracle, 它们可以精确地检测智能合约中的实际漏洞。第三, 我们在以太坊平台上对 6991 个现实世界的智能合约进行了系统模糊测试, ContractFuzzer 已识别出至少 459 个智能合约漏洞, 其中包括 DAO 错误和 Parity Wallet 错误。

其余部分的组织如下。在第 2 节中, 我们介绍了智能合约编程的初步知识, 并回顾了典型的智能合约漏洞。在第 3 节中, 我们将定义测试 oracle 以检测智能合约中的漏洞。然后, 在第 4 节中, 我们介绍 ContractFuzzer 框架的设计。之后, 我们在第 5 节中报告了一项全面的实验研究以评估 ContractFuzzer 的有效性, 然后在第 6 节中讨论了相关工作。最后, 在第 7 节中总结了我们的工作。

2 智能合约回顾

在本节中, 我们将简要回顾本工作中研究的智能合约的安全漏洞。

2.1 以太坊智能合约的基础

区块链的状态是从地址到帐户的映射。以太坊区块链平台不仅支持外部账户 (即由人拥有), 而且还支持智能合约账户[27], 这些账户在以太坊和通过代码管理的永久私有存储方面具有平衡。从概念上讲, 以太坊[22]可以看作是一个基于交易的状态机, 其状态在每笔交易上都会更新。此外, 交易的有效性由底层区块链平台的共识协议验证。交易是从一个帐户发送到另一帐户的消息。它可以包括二进制数据 (作为有效负载) 和以太。当目标是智能合约账户时, 将执行其代码并将有效负载作为输入数据提供。

智能合约的可执行代码是在基于堆栈的以太坊虚拟机 (EVM) 上运行的字节码。开发人员可以使用高级编程语言[16] Solidity 对智能合约进行编程, 然后将其编译为 EVM 字节码。创建后, 每笔交易都要收取一定数量的费用, 以支付其执行费用, 并避免恶意代码浪费以太坊资源。当合同执行过程中的天然气用尽时, 将触发天然气耗尽异常, 该异常将从交易意义上还原对帐户状态所做的所有修改。

2.2 漏洞的聪明合约在以太坊上

启用区块链的去中心化应用程序的安全漏洞可能发生在区块链级别, EVM 级别和智能合约级别。在这项工作中, 我们将重点放在智能合约的安全漏洞上, 我们将在本节中简要介绍这些漏洞。我们将遵循[1]和[9]的漏洞分类法。

无气发送。无气发送漏洞是由于这样的事实, 当使用发送时, 将调用接收者合同的后备功能, 但具有由 EVM 确定的固定气津贴。通常, 当发送量为非零时, 备用功能的气体限制为 2300。结果, 如果接收者合同具有昂贵的后备功能, 则以太坊的发送者将获得破产的例外。如果未检查并适当传播此类异常, 则恶意发件人可以在看似无辜的情况下错误地保留以太币。

异常疾病。异常异常是由于实体在异常处理方面不一致，这一事实取决于合同相互调用的方式。当合同调用另一个功能时，它可能会因不同类型的异常而失败。当发生此类异常时，处理机制由调用的方式确定。给定一连串的嵌套调用，其中每个调用都是对合同功能的直接调用，当发生异常时，所有交易将被还原（包括以太转移）。但是，对于一系列嵌套调用，其中至少一个调用是通过对地址（address.call（），address.delegatecall（）或 address.send（））的低级调用方法进行的，事务的回滚将仅在调用函数处停止并返回 false。从这一点出发，其他任何副作用都无法恢复，也不会传播任何掷球。在异常处理方面的这种不一致将使调用合同无法意识到执行期间发生的错误。

再入。可重入错误是由于以下事实造成的：某些功能并未设计为开发人员可重入。但是，恶意合约故意以可重入的方式（例如，通过后备功能）调用此类功能，因此可能会丢失以太币。著名的“ DAO”攻击就是利用此漏洞在以太坊方面造成了 6000 万美元的损失。

时间戳依赖性。时间戳依赖
当智能合约使用区块时间戳作为执行关键操作（例如，发送以太坊）的条件的一部分或作为生成随机数的熵的来源时，就会存在此漏洞。在像区块链这样的分布式系统中，矿工可以自由地在不到 900 秒的短时间间隔内设置一个区块的时间戳[24]。但是，如果智能合约基于时间戳传输以太币，则攻击者可以操纵区块时间戳来利用此漏洞。

块号依赖性。块号依赖性漏洞类似于时间戳依赖性。当智能合约使用 block.number 作为执行关键操作（例如发送以太坊）的条件的一部分或作为生成随机数的来源时，就会发生这种情况。实际上，block.timestamp 和 block.number 都是可由矿工操纵的变量，因此由于矿工的激励，它们不能用作熵的来源[30]。此外，由于 EVM 的执行机制或区块链的透明性，即使使用 block.number 作为参数用于随机数生成的 block.blockhash（）函数仍然容易受到攻击。

危险的 DelegateCall。委托调用与消息调用相同，除了目标地址上的代码是在调用合同的上下文中执行的[27]。这意味着合同可以在运行时从另一个地址动态加载代码，而存储仍然引用调用合同。这是在 Solidity 中实现“库”功能以重用代码的方式。但是，当委托调用的参数设置为 msg.data 时，攻击者可以使用函数的签名来制作 msg.data，以便攻击者可以使受害者签订合同以调用其提供的任何函数。第一轮奇偶钱包漏洞的爆发就证明了这一点[32]。如表 1 所示，电子钱包合约第 6 行包含一个以 msg.data 为参数的委托调用。这使得攻击者可以使用 Wallet 的数据调用_walletLibrary 的任何公共函数。因此，攻击者调用_walletLibrary 智能合约的 initWallet 函数（在第 10 行定义），并成为钱包合约的所有者。最后，他可以将钱包的以太币发送到自己的地址以完成攻击。这次攻击给平价钱包用户造成了 3000 万美元的损失。

表 1. 平价钱包合同中的危险委托电话

1	合约钱包{
2	function () payable {//后备功能
3	如果 (msg.value> 0)
4	存款 (msg.sender, msg.value) ;
5	否则 (msg.data.length> 0)
6	_walletLibrary.**delegatecall**(msg.data);
7	}
8	}
9	合约 WalletLibrary {
10	函数 initWallet (address [] _owners, uint _required, uint
11	initDaylimit(_daylimit);
12	initMultiowned(_owners, _required);
13	}
14	}

冻结谜。另一种易受攻击的合同是冷冻乙醚合同。这些合同可以接收以太币，并可以通过委托调用将以太币发送到其他地址。但是，它们本身不包含将以太币发送到其他地址的功能。换句话说，它们纯粹依靠其他合约的代码（通过委托调用）来发送以太币。当提供以太操纵代码的合同执行自杀或自毁操作时，主叫合同无法发送以太，并且所有以太被冻结。对 Parity 钱包漏洞的第二轮攻击仅仅是因为许多钱包合同只能依靠奇偶校验库来操纵其以太币。当通过初始化将奇偶库更改为合同，然后被黑客杀死时。依赖于奇偶校验库的钱包合约中的所有以太币均被冻结。

3 定义测试甲骨文智能合约的漏洞

在本节中，我们将定义测试 oracle，以检测智能合约中的每种类型的漏洞。

3.1 测试 Oracle 无气发送

在 EVM 中，send（）被实现为一种特殊的 call（）类型。因此，oracle GaslessSend 确保 EVM 中的调用确实是 send（）调用，并且 send（）调用在执行期间返回错误代码 ErrOutOfGas。要检查呼叫是否为 send（），我们验证呼叫的输入是否为 0 且呼叫的气体限制为 2300。

3.2 测试 Oracle 的异常情况

我们定义测试 oracle ExceptionDisorder 如下：对于源自根调用（或委托调用）的一连串嵌套调用（或委托调用），如果根调用未引发异常而其嵌套调用中的至少一个引发异常，则我们考虑电话

链包含异常障碍。换句话说，该异常未正确传播回根调用。

3.3 测试 Oracle 的可重入性

基于两个子 oracle 定义了测试 oracle reentrancy。第一个子 Oracle 是 ReentrancyCall，它检查函数调用 A 在源自调用 A 的调用链中是否出现多次。第二个子 Oracle 是 CallAgentWithValue，它检查三个条件：（1）有一个 call（）调用大于 0 的值（要转移的以太币数量）；（2）有足够的津贴用于被叫方执行复杂的代码执行（即，不发送或不发送）；（3）call（）的被调用方是我们工具提供的代理合同，而不是被测试合同指定的帐户。可重入测试预言定义为：

ReentrancyCall /¥ CallAgentWithValue

我们认为，当有一个调用通过一个调用链回调用到自己，并且该调用已通过 Call（）向以太子合约发送了以太气体给我们的代理合同提供了足够的汽油时，就会发生 Reentrancy 漏洞，从而使我们的代理合同可以再次执行重新进入的调用在其后备功能内。换而言之，我们的 ContractFuzzer 仅在能够成功对服务器发起再入攻击时才标记再入漏洞。
目标合同。我们将介绍代理合同的设计
及其攻击情形，请参见第 4.3 节。

3.4 测试 Oracle 的时间戳依赖性

测试 Oracle TimestampDependency 在三个子 Oracle 上定义。第一个子 Oracle 是 TimestampOp，它检查当前合同内的调用在执行过程中是否已调用 TIMESTAMP 操作码。第二个子 Oracle 是 SendCall，它检查该调用是否是将 ether 发送给其他合约的 send（）调用。第三个子预告片是 EtherTransfer，它检查 call（）的值（要转移的以太币的数量）是否大于 0。具体来说，TimestampDependency 定义为：

TimestampOp /¥ (SendCall ∨EtherTransfer)

总而言之，我们考虑 TimestampDependency
当前合同使用了块时间戳并且合同在执行期间转移了以太币时，会发生这种情况。

3.5 测试 Oracle 对块号的依赖性

测试 oracle BlockNumDependency 与 TimestampDependency 相似，不同之处在于它检查块编号的使用而不是块时间戳的使用。它也是基于 3 个子预定义的。第一个是 BlockNumberOp，它检查当前合同内的调用在执行过程中是否已调用 NUMBER 操作码。其他子 oracle 与 TimestampDependency 相同。并且 BlockNumDependency 定义为：

BlockNumOp /¥ (SendCall ∨EtherTransfer)

总而言之，我们认为 BlockNumDependency 发生在当前合约使用了区块编号并且合约在执行过程中转移了以太币时。

3.6 测试 Oracle 的危险 DelegateCall

测试 oracle DangerDelegateCall 检查在执行当前合同期间是否存在 deletegate 调用，以及是否从该合同的初始调用的输入（例如 msg.data）获得了委托调用所调用的函数。换句话说，测试 oracle 检查被测合同是否调用了委托调用，该委托调用的目标功能由潜在的攻击者提供。

3.7 测试 Oracle 的冻结醚合同

FreezingEther 的测试 oracle 检查合同是否可以接收以太币并在执行过程中使用了委托调用，但是当前合同本身内没有用于将以太币转移到其他地址的转移/发送/呼叫/自杀代码。换句话说，如果 FreezingEther 测试 oracle 在执行过程中其余额大于零，则将该合同标记为易受攻击的合同，但它无法通过自己的代码（即使用调用，转移和自杀）转移以太币。

4 智能合约模糊器

在本节中，我们首先概述一下 ContractFuzzer 工具。然后，我们继续详细介绍该工具的每个核心组件的设计。

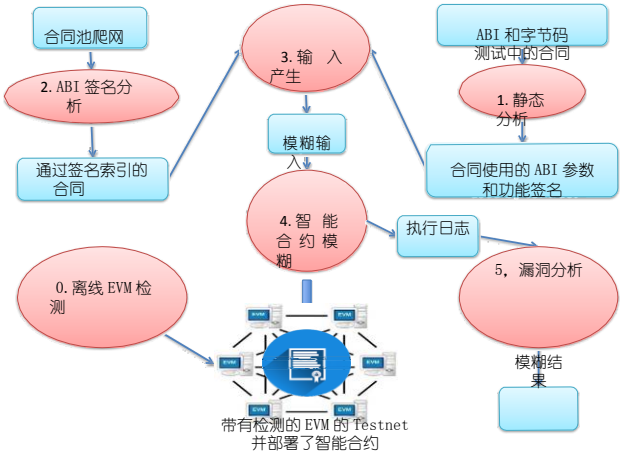

图 1 ContractFuzzer 工具概述

4.1 ContractFuzzer 概述

图 1 显示了 ContractFuzzer 的概述，描述了其工作流程，主要步骤是从 0 到 5 的数字。ContractFuzzer 工具包含一个离线 EVM 检测工具和一个在线模糊工具。步骤 0 中的脱机 EVM 检测过程负责检测 EVM，以便模糊测试工具可以监视智能合约的执行以提取信息以进行漏洞分析。我们还构建了一个 Web 爬网程序，以从 Etherscan [25]网站上提取以太坊平台上已部署的智能合约。我们的搜寻器将提取合同创建代码（智能合同的二进制文件），ABI 接口以及这些合同的构造函数参数。此外，我们还在以太坊测试网上部署了智能合约。部署的智能合约有两个目的：作为模糊测试的主题，以及作为使用合约地址作为参数的合约调用的输入。

在线模糊处理过程从第 1 步开始，其中 ContractFuzzer 工具将分析 ABI 接口和被测智能合约的字节码。然后，它将提取 ABI 函数的每个参数的数据类型，以及每个 ABI 函数中使用的函数的签名。在第 2 步中，该工具将对从以太坊爬网的所有智能合约执行 ABI 签名分析，然后通过它们支持的功能签名对智能合约进行索引。此步骤对于测试智能合约的交互至关重要。根据第 1 步和第 2 步的分析结果，该工具将生成符合 ABI 规范的有效模糊输入，以及在第 3 步中跨越有效性边界的变异输入。注意 ABI 不仅可以将通用数据类型指定为自变量，而且还可以将通用数据类型指定为自变量合同地址和功能选择器作为参数。从步骤 2 返回的索引智能合约用于生成以合约地址作为参数的 ABI 的输入。然后，在步骤 4 中，该工具将开始模糊处理，以使用随机函数调用将生成的输入针对 ABI 接口进行轰炸。最后，在步骤 5 中，该工具开始通过分析模糊测试期间生成的执行日志来检测安全漏洞。模糊测试过程将继续进行，直到可用的测试时间用完为止。当工具完成对所有受测试智能合约的模糊测试时，整个模糊测试活动便结束了。

4.2 智能合约的静态分析

合同池的 ABI 功能签名分析。ContractFuzzer 工具对智能合约池中的 ABI 执行静态分析，以提取这些合约支持的公共功能的签名。更具体地说，该工具基于 JSON 格式的每个智能合约的导出 ABI，提取在 ABI 中声明的所有功能签名。然后，我们为每个函数签名计算函数选择器，即函数签名的前四个字节 Keccak（SHA-3）哈希。最后，我们以功能选择器为键，并以具有相同功能选择器作为值的所有智能合约的地址向量构建映射。

被测智能合约的静态分析。以太坊智能合约的应用程序二进制接口（ABI）[15]是静态的并且是强类型的。我们解析合同的 ABI 接口的 JSON 格式，以提取其功能描述和每个参数的数据类型。可以根据文档[20]精确确定大多数数据类型的输入域。但是，地址数据类型本质上表示合同帐户或外部帐户的地址。当提供给函数的参数是合同地址时，函数可以使用调用函数与合同进行交互。因此，在为具有地址类型参数的 ABI 函数生成输入时，我们必须使用能够支持被测智能合约中调用的功能的智能合约的地址。

对于被测智能合约的给定 ABI 功能，我们如何有效确定可以与之交互的智能合约的子集？答案在于函数实现的字节码内的 call（）调用。call（）函数的参数的前四个字节对应于函数签名的前四个字节 Keccak（SHA-3）哈希，也称为函数选择器。

基于此观察，我们对智能合约字节码进行静态分析，以识别每个公共 ABI 功能的代码内使用的功能选择器，如表 2 所示。

算法的输入是二进制形式的智能合约，输出是映射，该映射将智能合约的每个 ABI 功能映射到其代码中使用的一组功能选择器。在第 2 行，该算法首先使用 EVM 工具 disam 将智能合约字节码反汇编为汇编代码。然后，该算法在第 4 行中以 funs 的形式提取其公共 ABI 函数集。从第 5 行到第 16 行，有一个循环迭代 funs 中的每个函数以获取其功能选择器集。在循环中，它首先提取函数的主体，然后从第 6 行到第 8 行找到函数的代码段集。从第 10 行到第 14 行，还有另一个循环迭代每个代码段的每一行，然后检查该行是否以“ PUSH4”操作码开头。如果是这样，它将拆分功能选择器。在第 15 和 16 行，算法使用选择器集设置 M [f]。最后，该算法返回整个地图 M。该算法在池中的每个智能合约上执行。

表 2. 函数选择器分析算法

	函数: FindFunctionSelectorForABI
	输入: bin, 二进制格式的智能合约
	输出: 映射 M 记录 ABI 的每个 ABI 功能中使用的功能选择器集合
1	**def** FindFunctionSelectorForABI(bin):
2	dasm_bin = disamble (bin) //反汇编二进制文件
3	//提取每个公共 ABI 函数签名
4	funs =extractFunction(dasm_bin)
5	有趣的 foreach f: //迭代每个公共函数
6	body = extractBody (f) //提取函数 f 的主体
7	//获取 body 的代码段
8	codeSegs = getCodeSegments(body)
9	选择器= set ()
10	代码段中的 foreach 段: //迭代代码段
11	段中的 foreach 行: //迭代每一行
12	如果 line.startswith (“ PUSH4”)
13	//提取一个函数选择器
14	selector.add(line.split()[1])
15	如果 len (selector) > 0: //确保选择器不为空
16	M [f] =选择器
17	return M //返回地图

如前几节所述，我们将具有相同功能选择器的所有智能合约的地址存储在映射中。因此，对于从功能选择器分析算法返回的每个功能选择器，我们搜索地图以查找所有支持该功能选择器作为 ABI 功能的智能合约地址。然后，我们将这些智能合约置于模糊状态下，用于智能合约的每个 ABI 功能的私有合约池中。在为 ABI 功能的地址数据类型生成模糊输入时，我们将使用该功能的私有智能合约池中的合约地址，这使智能合约的交互成为可能。

4.3 模糊输入生成

基于 ABI 接口的输入生成。输入生成算法负责为每个功能生成有效输入。我们使用不同的策略为固定大小的输入和非固定大小的输入生成输入。对于固定大小的输入类型，例如 INT <M>，UINT <M>，BYTES <M>和固定数组<type> [M]，我们首先通过随机构建一组值

在有效输入域内生成输入。然后，我们还基于静态分析为该类型的智能合约构建另一组种子输入，这些种子输入经常在智能合约中使用。然后，我们将这两个集合组合起来，以形成该类型的候选值集合。对于字节，字符串和<type> []之类的非固定大小的输入，我们分两步生成输入。我们首先随机生成一个正数作为长度。然后，我们从其输入域中随机选择元素。

ContractFuzzer 对要测试的智能合约的 ABI 中声明的每个功能执行模糊测试。因此，输入生成模块旨在针对每个功能生成一组候选输入。为此，输入生成模块对函数的每个参数进行迭代，以基于其输入域为每个参数生成 k 个候选。那么完整的输入集就是所有自变量的 k 个候选者的组合。最后，将输入集编码为字节码表示形式，以备调用。

输入产生可重入漏洞。与本工作中研究的其他漏洞不同，触发重新进入漏洞需要两个智能合约之间的交互调用。因此，我们不能仅通过从外部帐户调用合同来暴露这种漏洞。因此，我们需要生成一个可重入攻击方案，以尝试触发被测智能合约中的可重入漏洞。

表 3. 具有再入漏洞的合同

```
1  合同 BountyHunt {
2  ...
3  函数 ClaimBounty () preventTheft {
4    单位余额= bountyAmount [msg.sender];
5    如果 (msg.sender.call.value (balance) () ) {
6      totalBountyAmount-=余额;
7      bountyAmount[msg.sender] = 0;
8    }
9  }
10 }
11 合同 AttackerAgent {
12 ①  ...          ②          ③
13   功能 AgentCall (地址 contract_addr, bytes msg_data) {
14     call_contract_addr  = contract_addr;
15     call_msg_data = msg_data;
16     contract_addr.call(msg_data);
17   }
18   function () 应付款项{
19     call_contract_addr.call(call_msg_data);
20   }
21 }
```

因此，我们创建了一个 AttackerAgent 合约，以在可重入攻击场景下与被测智能合约的每个 ABI 功能进行交互。我们以 BountyHunt 智能合约的测试为例进行说明。在我们的实验中，ContractFuzzer 还成功检测到其中的重入漏洞。

如表 3 所示，ContractFuzzer 正在对 BountyHunt 智能合约的 ClaimBounty 函数进行模糊处理，以确定其是否包含可重入的 bug。为此，模糊器使用 AttackerAgent 尝试通过可重入攻击从中窃取以太币。首先，AttackerAgent 的 AgentCall 函数执行对 BountyHunt 智能合约的 ClaimBounty 函数的调用（第 14 至 16 行）以发起攻击（步骤 1）。在 ClaimBounty 函数中，BountyHunt 在第 7 行将对应的 bounryAccount 的值设置为 0 之前，在第 5 行发送以太调用，由于该调用未提供任何参数，因此它将调用该函数的回调函数（在第 18 行）。 AttackerAgent（步骤 2）。在回调函数中，AttackerAgent 可以再次调用 ClaimBounty () 函数作为可重入调用（步骤 3）。结果，BountyHunt 将再次将以太币发送给 AttackerAgent，直到其所有以太币都耗尽为止。借助 AttackerAgent，我们可以确保 ContractFuzzer 标记的每个重入漏洞确实可以利用。

4.4 检测 EVM 以收集测试 Oracle

根据我们提出的测试预言的定义，我们通常需要收集三种类型的信息。第一类信息是关于合同呼叫或委托呼叫的各种属性。第二种信息是关于执行期间调用的操作码的信息。第三类信息是合同执行期间的状态。

收集有关 Call / DelegateCall / Send 的信息。根据我们对智能合约漏洞的定义，所有这些都与处于模糊状态的智能合约的 Call / DelegateCall / Send 操作有关。Send 被视为 EVM 中的一种特殊类型的 Call，并且 DelegateCall 本质上与 Call 相同，除了在执行过程中使用上下文和调用者而不是被调用者的存储。因此，我们可以在检测过程中使用相同的 Call 数据结构为所有模型建模。

如表 4 所示，对于 Call，我们在执行期间收集以下信息。调用的这些属性对于支持大多数测试预言至关重要。

表 4 通话记录信息

呼叫者	发起呼叫的帐户的地址
被叫者	合同地址
功能	调用者调用的函数
输入	该函数的参数
值	发送给被调用方的以太币数量
加油站	允许使用汽油津贴
internal_calls	当前智能合约内的通话
opCode_stack	当前调用中执行的操作码

此外，为了支持测试 oracle Reentrancy 和 ExceptionDisorder，合同模糊处理程序需要有关模糊处理中当前合同之外的交叉合同调用的信息。因此，我们还记录了从初始调用开始的一系列调用，包括在当前合同和其他相关合同中进行的调用。

为了记录此类信息，我们在以太坊的 EVM 实现中对 EVM.Call () 和 EVM.DelegateCall () 函数进行了分析，以收集每个呼叫的与呼叫相关的信息。更具体地说，我们将 CALL opCode 推送到 opcode_stack 上，记录内部调用的信息，并将该调用追加到调用链上。

收集关于操作码的信息。一些测试 oracle，例如 TimestampDependency 和 BlockNumDependency 必须

检查某些操作码（例如 TIMESTAMP 和 NUMBER）的执行情况。此外，许多操作码可能会更改合同的状态，我们也需要记录该状态。为了记录操作码的执行，我们在以太坊的 Interpreter 实现中检测解释器.Run（）函数。在解释了所监视的每个操作码后，我们将其推入当前合同的操作码堆栈中。在这项工作中，我们从 129 条 EVM 指令中选择了 34 条用于检测，因为它们要么被我们的测试 Oracle 直接使用，要么可能导致合同状态发生变化，这对于安全性分析很有用。

4.5 漏洞分析与报告

漏洞分析和报告模块负责检测是否存在任何报告安全漏洞。当初始调用的调用堆栈为空时，EVM 中的检测模块将首先基于收集的检测信息检查那些子预言。然后，检测模型会将这些子 oracle 发送到侦听本地主机端口 8888 的 HTTP 服务器。服务器将收集子 oracle 的结果，然后对每个测试 oracle 的组合条件执行最终检查。

5 实验与结果分析

在本节中，我们介绍了实验的详细信息以及结果分析。

5.1 实验设置

我们使用台式电脑作为测试环境。该 PC 运行 Ubuntu 14.04 LTS，并配备了 Intel i5 8 核 CPU 和 16GB 内存。我们在 PC 中配置了两个泊坞窗，以帮助设置测试客户端和以太坊测试网。在运行测试客户端的 docker 中，我们使用了 node.js 运行时中的以太坊 Javascript API（web3.js 库）与 testnet docker 中的 geth 客户端进行交互。testnet docker 安装了 geth 客户端版本 1.7.0，然后我们还在该 docker 中创建了一个以太坊私有区块链，并以一个对等节点作为 testnet。我们将创世纪区块的初始挖掘难度设置为非常低的值，以便在模糊测试过程中可以快速进行交易确认。最后，我们在该测试网中部署了 6991 个智能合约进行实验。由于我们在此工作中仅关注智能合约漏洞，因此我们的测试网只包含一个对等方是很好的。

5.2 智能合约作为主题计划

在撰写本文时，在 Etherscan [25]上大约有 9960 个具有经过验证的源代码的不同智能合约。我们已经抓取了所有这些合约，并删除了一些智能合约，这些合约由于使用了无效的以太坊地址而无法部署在我们的测试网上。最后，将其余 6991 个智能合约用作我们的实验对象。我们使用合同创建代码（字节码），ABI 接口以及这些合同的构造函数参数作为 ContractFuzzer 的输入。我们选择带有源代码的智能合约，因为它们使我们更容易手动验证实验结果。但是 ContractFuzzer 只需要智能合约的字节码即可工作。

5.3 实验程序

ContractFuzzer 首先对每个合同执行静态分析，以为每个 ABI 准备私有合同池界面并提取 ABI 功能。利用静态分析结果和合同池，ContractFuzzer 继续生成输入。

对于处于模糊测试中的每个智能合约的每个 ABI 功能，ContractFuzzer 会使用三种类型的帐户对其进行调用。第一个是外部帐户，它是受测试合同的创建者。第二个是普通外部帐户，它与被测试合同没有关系。第三个是称为 AttackAgent 的合同帐户，可以在重入攻击情形下与目标合同进行交互。从这三种类型的帐户中的每一种，我们将被测试的智能合约称为两种模式：一种是带有以太坊转移，另一种不是。

对于每个 ABI 函数，如果包含参数，ContractFuzzer 将生成 k 个输入以调用它。否则，我们将简单地对其执行一个调用。结合 3 种类型的帐户和 call.value（）的 2 种选择，我们将为带参数的函数生成 6 * k 个调用，为无参数的函数生成 6（3 * 2）个调用。我们将 k 初始化为一个较大的值，以便 ContractFuzzer 可以选择多种候选调用。当对特定的智能合约进行模糊处理时，我们会将为其每个 ABI 函数生成的所有调用合并到对智能合约的调用池中。然后，ContractFuzzer 启动 HTTP 服务器以收集和分析测试 Oracle。对于每个智能合约，ContractFuzzer 会从其调用池中随机选择调用以执行模糊测试，从而模拟了智能合约功能的不同调用顺序。最后，服务器收集并分析结果以进行报告。经过大约 80 小时的模糊测试，我们停止了实验，直到结果逐渐收敛。

为了将我们的工具与最先进的智能合约验证工具进行比较，我们还使用了 Oyente 验证工具[9]来扫描 6991 个主题智能合约并将结果与 ContractFuzzer 进行比较。

5.4 实验结果与分析

在本节中，我们介绍我们的实验结果，然后针对每个研究问题进行详细的结果分析。

表 5 检测到的漏洞摘要

漏洞类型	数 漏洞	百分比	真实正利率
无气发送	138	2.06%	100%
异常疾病	36	0.54%	100%
再入	14	0.21%	100%
时间戳依赖性	152	2.27%	96.05%
块号依赖性	82	1.23%	96.34%
危险的委托电话	7	0.10%	100%
冷冻醚	30	0.45%	100%
总	459	/	/

检测到的漏洞摘要。在本节中，我们首先总结模糊测试活动的结果，如表 5 所示。这些列显示漏洞类型，为该类型找到的漏洞数量及其在测试的所有智能合约中的百分比。这些行代表每种漏洞类型的结果。

带有 Gasless Send 漏洞的智能合约数量为 138，约占测试的所有联系人的 2.06%。由于我们的无气发送测试 Oracle 会检查 EVM 中 ErrOutOfGas 错误的发生，因此我们的结果是精确的，不会产生误报。对于 Exception Disorder，ContractFuzzer 检测到 36 个易受攻击的智能合约，约占所有智能合约的 0.54%。我们还手动检查了检测到的每个智能合约，并确认它们都是真实的肯定。

ContractFuzzer 检测到的 Reentrancy 漏洞数量为 14，大约占该实验研究的所有智能联系人的 0.21%。我们手动检查了这些情况，并确认没有报告误报。还检测到臭名昭著的 DAO 错误。

对于时间戳依赖和块编号依赖，ContractFuzzer 分别检测到 152 和 82 个易受攻击的智能合约。我们手动检查了这些智能合约以进行确认。我们发现在标记为时间戳依赖的 152 个智能合约中有 6 个为误报，在标记为 Block Number Dependency 的 82 个智能合约中有 3 个为误报。因此，真实阳性率分别为 96.05%和 96.34%。误报案例的原因是由于我们针对这两种漏洞的测试 Oracle 定义不准确。确实，在两种情况下的测试 oracle 定义中，我们都检查了操作码（TIMESTAMP 和 NUMBER）的使用以及在测试的 ABI 函数中以太传输调用的使用。我们尚未检查在读取 TIMESTAMP 和 NUMBER 以及在计算以太转移条件时使用它们之间是否存在数据流取消使用链。但是，记录此类信息可能会涉及通过测试智能合约的检测来进行昂贵的数据流跟踪。考虑到较高的真实阳性率，我们认为 ContractFuzzer 的解决方案是一种经济高效的折衷方案。

我们检测到 7 个“危险的委托呼叫”漏洞，我们确认均为真实。这些智能合约都使用初始调用者的输入来提取委托调用要调用的函数。还可以检测到导致第一轮奇偶校验错误的智能合约。对于“冷冻乙醚”漏洞，我们检测到 30 个案例。我们检查了这些智能合约，并确认它们确实没有办法用自己的代码发送以太币，并且它们只能通过其他合约间接转让以太币。总计，ContractFuzzer 精确地检测到 459 个智能合约，它们具有很高的真实肯定率。

表 6 合同模糊测试与 Oyente 的比较

漏洞类型	ContractFuzzer			奥恩特		
统计	没	FP	n	没	FP	n
时间戳依赖性	152	6	95	273	70	44
再入	14	0	1	43	28	0

与 Oyente 验证工具的比较。我们还将 ContractFuzzer 与 Oyente 验证工具[9] [29]进行了比较，以发现某些常见类型的漏洞。公开发布的 Oyente 工具[29]可以检测四种类型的漏洞。其中之一是呼叫堆栈大小限制漏洞，它不再相关，因为在以太坊 EIP150 硬叉中已经解决了该漏洞。其他 3 种 Oyente 工具检测到的漏洞是事务排序依赖关系，时间戳依赖关系和重入。事务排序依存关系的测试需要事务确认过程的操作，ContractFuzzer 尚不支持此操作。因此，ContractFuzzer 和 Oyente 只能检测到时间戳依赖和重入漏洞。通常，如果我们添加事务排序依赖项漏洞，ContractFuzzer 可以检测到 8 种类型的漏洞中的 7 种，而 Oyente 只能检测到 8 种类型的漏洞中的 3 种。因此，ContractFuzzer 可以检测更多类型的漏洞。

如表 6 所示，对于时间戳依赖性，ContractFuzzer 检测到 152 个漏洞，Oyente 检测到 273 个漏洞。我们手动检查了这些智能合约，发现 ContractFuzzer 和 Oyente 都有误报和误报。ContractFuzzer 的误报有两个原因。第一个是一些合约在其代码中对特定时间（即众筹开始日期）进行了硬编码，并将区块时间戳与之进行比较。如表 7 所示，BDSM_Crowdsale 智能合约的代码段指定了两个固定日期作为 ICO 的开始日期和结束日期（第 3 行和第 4 行）。然后，合同将区块时间戳与 ICO 开始日期进行比较，以确定是否执行退款（第 6 行和第 7 行）。显然，智能合约依赖于时间戳，因为它使用当前的块时间戳来决定是否执行以太坊传输。但是，由于特定日期已经过去，因此条件将始终失败。结果，在执行期间将不会发生以太坊转移，并且 ContractFuzzer 不会认为它与时间戳有关。ContractFuzzer 错误否定的第二个原因是，在有限的测试时间内很难触发某些条件。例如，一个合约在触发依赖于时间戳的以太转移之前，需要功能的特定调用模式。我们可以使用不同的函数调用序列执行更广泛的模糊测试，以改善这种情况。

表 7. ContractFuzzer 的假阴性案例

```
1  合同 BDSM_Crowdsale {
2    ...
3    uint public startICO_20_December = 1513728060; //2017.12.20
4    uint public stopICO_20_March = 1521504060; //2018.3.20
5    功能 () 应付款项{
6      如果 (现在<startICO_20_December) {
7        msg.sender.transfer(msg.value);
8      }
9    ...
```

Oyente 工具的假阴性主要是由于难以对符号运算进行符号分析。许多易受攻击的智能合约都利用区块时间戳或区块编号作为具有加密功能的随机种子生成的来源。例如，表 8 的 Bomb 合同的代码片段调用了诸如块哈希和 keccak256 之类的加密函数，并以块号和时间戳作为输入（第 3 行）。然后在传输以太之前检查结果是否等于 1（第 4 行）。该代码几乎是不可能进行符号分析的，因为它基本上是

要求将 1 取反的输入，这正是密码功能设计的难点。结果，Oyente 可能会错过以此模式编写的漏洞。Oyente 工具也有一些误报的情况。经过人工检查，我们发现时间戳从未用于计算传输以太的路径条件。

对于可重入漏洞，ContractFuzzer 检测到 14 个漏洞，Oyente 检测到 43 个漏洞。通过手动检查，我们确认 ContractFuzzer 检测到的所有 14 个漏洞都是真实的肯定。但是，对于 ContractFuzzer，缺少一个易受攻击的智能合约（即误报）。我们检查了这些智能合约，发现某些智能合约的脆弱功能必须在转移以太坊之前执行复杂的条件检查。但是，ContracFuzzer 很难触发这些条件。

表 8. Oyente 的假阴性案例

```
1  函数 buy (uint8 _bomb) 公共应付款项{
2  ...
3  int _random =
   uint(keccak256(block.blockhash(block.number1),msg.gas,
   tx.gasprice,block.timestamp))%bomb.chance + 1;
4  if(_random ==1){
5     bomb.owner.transfer(...)
6     ceoAddress.transfer(...)
7  }}
```

对于 Oyente 工具，在我们的测试 Oracle 定义中，被标记为可重入的 43 个智能合约中有 28 个是误报。我们将 28 例假阳性病例分为 3 种类型。第一种类型是智能合约，它使用 send () 和 transfer () 操作以及有限的燃气津贴，这使得被调用方的回调函数没有足够的燃气来再次执行重入呼叫。第二种智能合约的易受攻击的功能会严格检查其调用者是否是在合同创建期间指定的智能合约的所有者。由于外部帐户无法调用包含以太坊转移的功能，因此也无法进行重入攻击。最后，第三种智能合约只能将以太币传输到硬编码地址。恶意合同无法获取以太币或触发可重入呼叫。因此，根据我们的定义，这 28 个案例是误报案例：再入合同永远不能通过外部合同触发。ContractFuzzer 没有报告那些误报的情况，因为它同时检查了可重入呼叫和以太币到外部帐户的转移。

总而言之，与 Oyente 相比，ContractFuzzer 的两种脆弱性假阳性率要低得多。但是，对于 Timestamp Dependency，ContractFuzzer 的误报率很高，我们可以使用更全面的输入生成方案来进行改进。

5.5 攻击脆弱智能合约的案例研究

在本节中，我们将介绍一些有关 ContractFuzzer 检测到的可利用智能合约的案例研究。

错误地持有投资者的以太币。表 9 中的 CrowdSalePreICO 是一种恶意的智能合约，利用异常异常漏洞错误地持有投资者的以太币。第 3 行的函数是智能合约的回调函数。ContractFuzzer 将使用空输入和一些值来调用 CrowdSalePreICO，以充当投资者。然后，回调函数将检查收到的以太币并将其添加到总存款中（第 5 至 8 行）。但是，如果总存款超过众筹限额，则多余的以太币将退还给投资者（第 9 至 11 行）。但是，第 11 行的发送可能会失败。但是 CrowdSalePreICO 智能合约不会检查和处理错误。结果，募捐人将错误地保留投资者的过多以太。

表 9. ICO 合同错误地持有投资者的谜

```
1  合同 CrowdSalePreICO {
2     ...
3  function() payable stopInEmergency onlyStarted notFinalized{
4     ...
5     uint 贡献= msg.value;
6  如果 (safeAdd (totalDepositedEthers, msg.value) >
   hardCapAmount)
7  贡献= safeSub (hardCapAmount, totalDepositedEthers) ;
8     }
9  uint 过量= safeSub (msg.value, 贡献) ;
10    如果 (超过> 0) {
11       msg.sender.send(excess);
12    }
13    }
14 }
```

操作时间戳以赢得老虎机。如表 10 所示，合同 SlotMachine () 是一种智能合同，它利用区块时间戳作为随机数来确定老虎机游戏的获胜者。

表 10. 可利用的老虎机智能合约

```
1  合同 SlotMachine {
2     ...
3  功能 () {
4     uint nr = now;//现在是块时间戳
5     uint y = nr & 3;
6     ...
7  if(y==1){wins[1]++;win=(msg.value*2)+(msg.value/2);}
8     收益+ = int (信息价值) ;
9  if(win > 0){
10    bool res = msg.sender.send(win);
11    收益-= int (win) ;
12 }}}
```

在第 4 行，将块时间戳读入 nr。然后使用 nr 计算获胜（第 5 至 7 行）。然后使用获胜来确定是否将奖励发送给函数的调用者（第 9 至 11 行）。如果以太坊区块链的矿工参加了老虎机游戏，他可以操纵区块时间戳的值（即现在）以利于他/她的兴趣。这种攻击存在于

所有利用区块时间戳和区块编号来确定以太币转让条件的智能合约。

依靠硬编码库传输以太。如表 11 所示，这是受第二轮奇偶校验错误影响的钱包合同。在电子钱包智能合约中，它可以调用 walletLibrary 的代码来操纵其帐户或执行其他操作（第 7 行和第 10 行）。但是，电子钱包代码不包含用于转移以太币的电话/转移/自杀。更糟糕的是，walletLibrary 被定义为硬编码地址。当 walletLibrary 更改为智能合约帐户并被终止后，电子钱包合约无法发送以太币，其以太币将被冻结。在 2017 年 11 月平价漏洞攻击期间，2.8 亿美元的以太币被冻结在这样的 Wallet 智能合约账户中。

表 11. 被奇偶校验错误冻结的电子钱包智能合约

```
1  合约钱包是 WalletEvents {
2    ...
3    function () 应付帐款{
4      如果 (msg.value> 0)
5        存款 (msg.sender, msg.value) ;
6      否则 (msg.data.length> 0)
7        _walletLibrary.delegatecall(msg.data);
8    }
9    函数 hasConfirmed (bytes32 _operation, 地址_owner) …{
10     返回_walletLibrary.delegatecall (msg.data) ;
11   }
12   地址常量_walletLibrary =
     0xcafecafecafecafecafecafecafecafecafecaf
13 }
```

6 相关工作

在本节中，我们简要回顾有关智能合约漏洞和攻击的相关工作，以及用于检测此类安全错误的安全测试和验证技术。

6.1 智能合约错误和漏洞

Delmolino 等。[4]表明，即使是很小的智能合约，也可能存在很多逻辑问题。他们列出了一些常见的逻辑问题，例如合同从未将以太币退还给其发送者，以及未加密的数据泄露了隐私。Miller [10]审核了智能合约的源代码，并报告了以太坊上的调用堆栈溢出错误。在 2016 年遭受持续攻击后，该问题通过以太坊上的硬分叉解决。Atzei [1]系统地调查了对以太坊智能合约的安全攻击。他们根据其特征提供了智能合约漏洞的分类法。

6.2 智能合约安全

Fillâtre[6]提出了用于程序验证的工具 Why3，现在可以在 Solidity Web IDE [9]中将其作为正式的验证后端使用。在 IDE 中对智能合约进行编程时，该工具可以帮助检查整数数组的排列，数学运算的溢出以及零错误的除法。

Bhargavan [3]设法研究了智能合约的安全性和 Solidity 编译器的可靠性。他们设计了一个工具套件，可以转换 Solidity 源代码和将 EVM 字节码分别转换为 F * [3]程序。Luu 等人[9]设计了 Oyente，这是一种用于智能合约的符号验证工具。Oyente 构建智能合约的控制流程图，然后在控制流程图上执行符号执行，同时检查是否存在任何易受攻击的模式。

Nikolic 等人[12]设计了 MAIAN，这是一种符号执行工具，用于推理跟踪属性以检测易受攻击的智能合约。它基于跟踪属性指定了三个典型的智能合约漏洞。MAIAN 可以通过象征性执行有效地检测贪婪，浪子和自杀契约。与他们的工作不同，ContractFuzzer 工具执行模糊测试和运行时监视以检测执行期间发生的漏洞，从而可以减少误报。平井[8]使用 Isabelle / HOL 工具来验证名为 Deed 的智能合约，该合约是以太坊名称服务实现的一部分。具体来说，该工作验证了只有 Deed 的所有者才能减少其余额的预言。此外，他们还发现在验证过程中对 EVM 实施的测试很差。Echidna [21]是一个智能合约模糊器，在测试人员的单元测试中定义了预言。它还可以生成模糊智能合约的输入。但是，它不提供直接 API 支持智能合约的安全性测试。

6.3 漏洞检测的模糊技术

许多黑盒模糊测试器都是基于语法的，例如 SPIKE [2]和 Peach [5]。Hanford [7]和 Purdom [13]在 1970 年代开始研究基于语法的测试用例的生成。他们证明了基于语法的模糊器可以有效地检测正在测试的应用程序中的漏洞。

7 结论

随着区块链的普及和智能合约技术的发展，数以百万计的智能合约已部署在区块链平台上，以实现去中心化应用程序的构建。但是，智能合约的安全漏洞对其未来构成了巨大威胁。在这项工作中，我们提出了 ContractFuzzer，这是一个精确而全面的模糊测试框架，可以检测 7 种以太坊智能合约漏洞。我们对 6991 个现实世界智能合约的实验表明，ContractFuzzer 的输入生成和测试 Oracle 分析策略可以非常高精度地有效触发和检测安全漏洞。在测试的 6991 个智能合约中，我们的工具报告了 459 个漏洞，其中包括臭名昭著的 DAO 错误和 Parity Wallet 错误。与最先进的安全验证工具 Oyente 相比，ContractFuzzer 不仅可以检测更多类型的漏洞，而且其误报率要低得多。

在以后的工作中，为了减少误报，我们可能会针对这些类型的智能合约错误研究漏洞利用模式，这可能会指导我们生成更有效的漏洞触发输入。我们还将扩展我们的工具，以检测与 EVM 或基础区块链平台相关的更多类型的智能合约漏洞。最后，我们还将把工作推广到其他智能合约平台的安全测试。

参考资料

[1] N. Atzei, M. Bartoletti, T. Cimoli.以太坊智能合约攻击调查。在第六届国际安全与信任原则会议的会议记录中，第 164-186 页，柏林，海德堡，2017 年。

[2] D. Aitel. 穗 。 http://immunityinc.com/resources-freesoftware.shtml，2002 年。

[3] K. Bhargavan，N。Swamy，S。Zanella-Béguelin 等。正式验证智能合约。2016 年 ACM 研讨会论文集，第 91-96 页，2016 年。

[4] K. Delmolino，M。Arnett，A Kosba 等。逐步创建安全的智能合约：加密货币实验室的经验教训和见解。在《第 16 届金融密码学和数据安全论文集》中，第 79-94 页，柏林，海德堡，2016 年。

[5] M. Eddington.桃子。http://www.peachfuzzer.com,2004.

[6] JC Filliâtre，A. Paskevich.Why3-程序遇到问题的地方。在第 20 届 ACM SIGPLAN 编程语言设计与实现会议记录中，第 125-128 页，柏林，海德堡，2013 年。

[7]汉福德（KV Hanford）。自动生成测试用例。IBM 系统杂志，9（4）：242-257，1970。

[8] Y.平井。以太坊名称服务中契约合同的正式验证。http://yoichihirai.com/deed.pdf,2016.

[9] L. Luu，Dh Chu，H。Olickel 等。使智能合约更智能。在第 23 届 ACM SIGSAC 计算机和通信安全会议论文集，第 254-269 页，奥地利维也纳，2016 年。

[10]米勒。B. Warner，N。Wilcox 等。天然气经济学。http://github.com /LeastAuthority/ethereumanalyses/blob/master/GasEcon.md，2015 年。

[11]中本聪。比特币：点对点的电子现金系统。https://bitcoin.org/bitcoin.pdf，2009 年。

[12] I. Nikolic，A。Kolluri，I。Sergey 等。大规模查找贪婪，败家子和自杀合同。https://arxiv.org/pdf/1802.06038，2018 年。

[13] P. Purdom.用于测试解析器的句子生成器。位数值数学，12（3）：366-375，1972。

[14] N. Swammy，C。Keller，A。Rastogi 等。F *中的从属类型和多单峰效应。在第 43 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集，第 256-270 页，圣彼得堡，佛罗里达州，2016 年。

[15]阿比 的 以太坊 聪明 合同。https://github.com/ethereum/wiki/wiki/Ethereum-ContractABI。上次访问 2018。

[16] 以太坊智能合约的 ABI 规范。http://solidity.readthedocs.io/en/v0.4.21/abi-spec.html#.

[17]分析 的 的 道 利用。http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.上次访问时间，2018 年。

[18]宣布将为 EIP150 燃气成本进行艰苦的工作 变化，https://blog.ethereum.org/2016/10/13/announcement-imminent-hard-fork-eip150-gas-cost-changes/.

[19]加密货币市场资本。https://coinmarketcap.com.上次访问时间，2018 年。

[20]数据 种类 在 坚固性。https://solidity.readthedocs.io/en/develop/types.html#.上次访问 2018。

[21]针 chi。https://github.com/trailofbits/echidna。

[22]以太坊 白色 纸 。https://github.com/ethereum/wiki/wiki/White-Paper 。上次访问时间，2018 年。

[23]以太坊 黄色 纸 。https://ethereum.github.io/yellowpaper/paper.pdf.上次访问时间，2018 年。

[24]以太坊 块 验证 算法。https://github.com/ethereum/wiki/wiki/Block-Protocol2.0#block-validation-algorithm。

[25] Etherscan. 的 以太坊 块 资源管理器。https://etherscan.io/.上次访问时间，2018 年。

[26]正式 验证 对于 坚固性 合同。http://forum.ethereum.org/discussion//3779/formal- 实体合同核查，2015 年。

[27]简介 至 聪明 合同。http://solidity.readthedocs.io/en/v0.4.21/introduction-to-smart-contracts.html.上次访问时间，2018 年。

[28]编号 的 聪明 合约 上 以太坊。https://etherscan.io/accounts/c.上次访问时间，2018 年。

[29] Oyente 分析 工具 对于 聪明 合同。https://github.com/melonproject/oyente。上次访问时间，2018 年。

[30]预测以太坊智能合约中的随机成员。https://blog.positive.com/predicting-random-numbers-inethereum-smart-contracts-e5358c6b8620。上次访问时间，2018 年。

[31]聪明 合约 与 已验证 资源 码。https://etherscan.io/contractsVerified.上次访问时间，2018 年

[32]平价 钱包 哈克 解 释 。https://blog.zeppelin.solutions/on-the-parity-wallet-multisighack-405a8c12e8f7.上次访问时间，2018 年

[33]被奇偶错误冻结的电子钱包智能合约。https://github.com/paritytech/parity/blob/4d08e7b0aec46443 bf26547b17d10cb302672835/js/src/contracts/snippets/enhanc ed-wallet.sol.上次访问时间，2018 年。

[34]可升级 聪明 合 同 。https://ethereum.stackexchange.com/questions/2404/upgrade able-smart-contracts.上次访问时间，2018 年。