DC-Hunter: 一种基于字节码匹配的危险智能合约检测方案

韩松明, 梁　彬, 黄建军, 石文昌

中国人民大学信息学院 北京 中国 100872

摘要　近年来, 智能合约中的漏洞检测任务已受到越来越多的关注。然而, 缺少源代码和完备的检测特征限制了检测的效果。在本文中, 我们提出了 DC-Hunter: 一种基于字节码匹配的智能合约漏洞检测方案。它可以通过已知的漏洞合约找到类似的漏洞合约, 并且可以直接应用于现实世界中的智能合约, 无需源码和预先定义的漏洞特征。为了让提出的方法更加切实可行, 我们应用程序切片来降低无关代码的影响, 通过规范化减少编译器版本带来的差异, 并使用图嵌入算法来捕捉函数的结构信息, 从而显著减少误报和漏报。此外, 借助 DC-Hunter 我们揭露了一种新型的危险合约。我们发现有一些合约是伪漏洞合约, 专门用于诱骗他人尝试进行攻击, 从而窃取攻击者的以太币, 这种合约称为“蜜罐合约”。我们实现了 DC-Hunter 的原型, 并将其应用于现实世界的智能合约, 共有 183 份危险的合约被报出并确认, 其中包括 160 份漏洞合约和 23 份蜜罐合约。

关键词　字节码匹配; 切片; 规范化; 图嵌入; 蜜罐

中图法分类号　TP311　DOI 号 10.19363/J.cnki.cn10-1380/tn.2020.05.08

DC-Hunter: Detecting Dangerous Smart Contracts via Bytecode Matching

HAN Songming, LIANG Bin, HUANG Jianjun, SHI Wenchang

School of Information, Renmin University of China, Beijing 100872, China

Abstract　In recent years, detecting vulnerabilities in smart contracts has become a critical task. However, the detection performance is subject to lack of source code and comprehensive detection signatures. In this paper we present a smart contract detection method based on bytecode matching, called DC-Hunter. It can effectively find vulnerable smart contracts by retrieving the analogues of known vulnerable contracts, and can be directly applied to the real-world smart contracts without requiring source code and predefined signatures. To make the proposed method more practicable, we utilize program slicing to mitigate the impact of irrelevant code, perform normalization to reduce the differences caused by compiler versions, and use graph embedding network to capture the structural information of functions, so that false positives and false negatives are significantly pruned. Besides, we expose a new type of dangerous contract with help of DC-Hunter. We find that there are some pseudo-vulnerable contracts specially designed for seducing people into attacking them to steal their ether, which are called honeypot contracts. We implement DC-Hunter and apply it to real-world smart contracts. 183 dangerous contracts are reported and confirmed, including 160 vulnerable ones and 23 honeypot contracts.

Key words　bytecode matching; slicing; normalization; graph embedding; honeypot

1　引言

以太坊是一个公共的区块链平台, 智能合约则是一种主要以 Solidity 脚本语言编写的程序, 可部署并运行于以太坊平台上, 在区块链应用中起着至关重要的作用。然而, 开发者缺乏安全意识等因素会导致智能合约中存在漏洞, 这些漏洞进而可能导致巨大的直接经济损失。例如, 智能合约 theDAO 和 BeautyChain 中的漏洞已导致损失了价值超过 10 亿美元的数字货币[1-2]。此外, 由于区块链的性质, 智能合约一旦部署, 便很难对其进行修补。因此, 智能合约的漏洞检测工作正受到越来越多的关注。

但是到目前为止, 还不存在一个完备的智能合约漏洞特征库。实际上, 完全收集所有的漏洞逻辑并归纳相应的特征是难以实现的。对此, 有一种直观而可行的解决方案: 将已知漏洞直接当作一种特殊的

通讯作者: 梁彬, 博士, 教授, Email: liangb@ruc.edu.cn。

本课题得到国家自然科学基金(No.U1836209, No. 61802413)资助。

收稿日期: 2020-04-04; 修改日期: 2020-04-30; 定稿日期: 2020-05-08

特征, 与待检测代码一同转化为向量等易于比较的形式, 并基于此进行匹配以发现与其相似的实现。如果有二者的相似度超过设定的阈值, 则认为其中的待测代码是可疑对象, 并对其进行进一步审计。值得一提的是, 目前只有少数智能合约是开源的。根据我们的统计, 前 150 万个部署在区块链上的智能合约中, 只有 32,499 个合约(约 2%)在以太坊浏览器 Etherscan[3]上公开了源代码。此外, 近期 Etherscan 对网站功能进行了调整, 此前可直接检索到所有已公开的合约源代码, 而现在只能检索到最新的 500 份合约源代码, 这样的调整进一步提升了合约源代码获取的难度。对于绝大部分的闭源合约, 我们只能对其字节码进行分析。为此, 我们提出了 DC-Hunter: 一种基于字节码匹配的智能合约漏洞检测方案, 通过计算待测合约与已知漏洞合约之间的相似性, 高效地检测潜在的漏洞合约。

值得注意的是, 尽管基于匹配的检测方法在其他领域中已被证明是有效的[4-6], 其应用于智能合约的字节码时仍将面临一些特殊的挑战:

其一, 与传统程序相比, 智能合约会显得更为同质, 这使得匹配工作更容易受到与漏洞逻辑无关的代码段, 也就是所谓“噪声”的影响。在如图 1 所示的漏洞合约中, 仅第 3、4 行代码与漏洞逻辑相关, 其余均为与漏洞逻辑无关的噪声代码。当用其作为“种子”进行匹配时, 就有可能错误地捕获许多仅与噪声代码拥有较高相似度的合约, 从而导致大量的误报。

```
1 function creditEqually(address[] users, uint256 value) public
      onlyMaster returns (bool) {
2     uint256 balance = balances[msg.sender];
3     uint256 totalValue = users.length * value;
4     require(totalValue <= balance);
5     balances[msg.sender] = balance - totalValue;
6     /* transfer things, 15 loc are omitted */
7 }
```

图 1 BeerCoin 合约[7]

Figure 1 Contract BeerCoin[7]

其二, 智能合约的源代码由 Solidity 编译器编译为字节码, 而目前有几十个版本的 Solidity 编译器可供使用[8]。不同版本的编译器中, 字节码生成的方式有很大的差别。图 2 显示了 BecToken 合约中的关键漏洞语句经由编译器 v0.4.11 和 v0.4.25 编译生成的字节码。可以发现, 即使是完全相同的源代码, 由不同版本的编译器生成的字节码也有很大差异。实际上, 编译器的多样性在许多方面都给匹配工作带来了阻碍——即便潜在漏洞合约与已知的漏洞合约几乎完全相同, 也很有可能因编译器版本不同而造成字节码上的差异, 使得它们之间的相似度降低, 最终导致漏报。

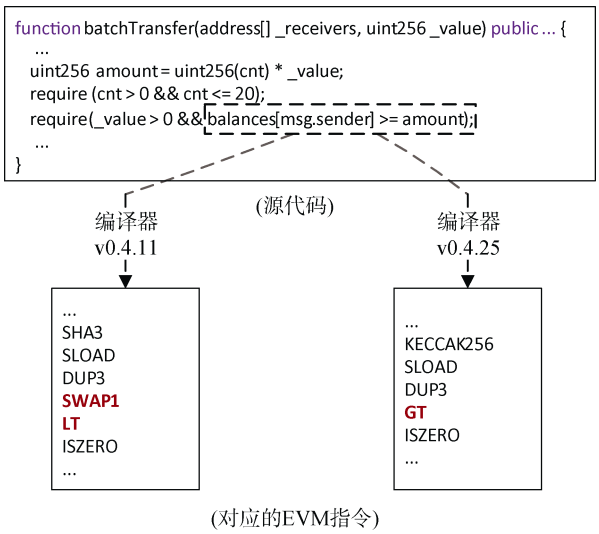


图 2 来自不同编译器的字节码

Figure 2 Bytecode from different compilers

其三, 在代码匹配的过程中, 为了方便进行相似性度量, 代码将会被转换为可定量、可测量、可比较的形式。例如, 在 Yamaguchi 等人[5-6]的工作中, 每个函数的特征信息会通过 Bag-of-Feature 的方式编码为向量。虽然这项工作取得了良好的效果, 但是它缺乏对函数结构信息的考虑, 在智能合约领域, 这可能导致漏报。例如, 图 3 与图 4 均为带有整数溢出漏洞的转账函数, 虽然它们的溢出方式并不相同, 但函数的部分结构较为相似。如果忽略函数的结构信息, 将很难捕获它们之间的相似性。

为了解决上述挑战, 我们对程序进行切片及规范化, 并将规范化后的切片通过图嵌入网络映射为向量, 进而进行匹配。具体地说, 首先我们利用程序切片技术减少噪声的影响。在进行匹配之前, 合约的字节码会被切成多个切片(Slice), 每个切片描述了一个数据所经历过的指令。由此, 可以有效地将噪声代码从最有可能体现漏洞核心逻辑的切片中排除, 代码匹配可以在相对“纯净的”代码集上进行, 从而获得更好的效果。其次, 我们将从数值规范化、参数顺序调整和移除无关紧要的指令等三个角度对获得的代码切片进行规范化, 从而使得在源码层面相似的合约在字节码层面尽可能地保持相似, 提高匹配的准确性。第三, 为了捕获代码的结构信息, 我们将每个切片都视为一张图, 并通过图嵌入网络来计算这些切片的嵌入向量, 从而能够匹配实现方式不同, 但代码逻辑相似的合约。

```
function multiTransfer(address[] destinations, uint[] tokens)
      public returns (bool success){
    uint totalTokensToTransfer = 0;
    for (uint8 i = 0; i < destinations.length; i++){
        assert(tokens[i] > 0);
        totalTokensToTransfer += tokens[i];
    }
    assert (balances[msg.sender] > totalTokensToTransfer);
    balances[msg.sender] =
          balances[msg.sender].sub(totalTokensToTransfer);
    for (i = 0; i < destinations.length; i++){
        balances[destinations[i]] =
              balances[destinations[i]].add(tokens[i]);
    }
    return true;
}
```

图 3　Ammbr 合约[9]

Figure 3　Contract Ammbr[9]

```
function batchTransfer(address[] _receivers, uint256 _value)
      public whenNotPaused returns (bool) {
    uint cnt = _receivers.length;
    uint256 amount = uint256(cnt) * _value;
    require(cnt > 0 && cnt <= 20);
    require(_value > 0 && balances[msg.sender] >= amount);
    balances[msg.sender] = balances[msg.sender].sub(amount);
    for (uint i = 0; i < cnt; i++) {
        balances[_receivers[i]] =
              balances[_receivers[i]].add(_value);
    }
    return true;
}
```

图 4　BecToken 合约[10]

Figure 4　Contract BecToken[10]

我们实现了 DC-Hunter 的原型, 并使用它检测了超 200 万份合约的字节码, 其中包括 3 万余个开源的合约。结果表明, DC-Hunter 可以有效地提高代码匹配的精度, 并能成功检测出未知的漏洞合约。与直接匹配相比, 我们的方法可以通过程序切片技术减少 90%以上的误报。此外, 应用规范化技术也使得检测出的漏洞数量有显着的增加。最终, 我们成功找到了 160 个漏洞合约。

值得一提的是, 我们在审计结果的过程中有一些非常意外而又有趣的发现。在 DC-Hunter 捕获的合约中有这么一部分合约, 它们包含已知的漏洞模式, 但并不是真正的漏洞合约, 而是一种特殊的“蜜罐”合约。通过观察, 我们发现这种“蜜罐”合约的代码中会包含一些较为有名的漏洞模式, 从而伪装成可被利用的漏洞合约, 让攻击者认为可以在支付少量的以太币后, 通过利用漏洞来窃取该合约持有的大量以太币。然而, 当攻击者尝试攻击时, 合约中某些隐蔽的逻辑会使得攻击者无法成功通过漏洞窃取合约中的以太币, 反而会损失掉已为进行攻击而支付的以太币。尽管在原始环境中蜜罐合约无法被成功攻击, 但它们也同样也具有危险性。在本文中, 我们把这两类合约统称为危险合约。为了证明所检测到蜜罐合约的危害性, 我们将其部署到私有链上, 并尝试进行攻击。实验证明, 所有的蜜罐合约具有锁定攻击者的以太币的功能, 并且被锁定的以太币能被合约的所有者取走。到目前为止, 我们已经在 DC-Hunter 的帮助下发现了 23 个蜜罐合约。

2　背景介绍

本节中我们将简述工作的背景, 包括一些智能合约的漏洞类型以及以太坊虚拟机(Ethereum Virtual Machine, EVM)的数据处理机制。

2.1　漏洞类型

在本文中, 我们主要关注以下 5 类漏洞: 整数溢出、重入、不良随机源、访问控制和遗漏异常处理, 这些漏洞的产生均与对不安全数据的不恰当操作有关。下面分别对其进行简要介绍。

1) 整数溢出: 图 4 是漏洞 BecToken 合约中存在整数溢出漏洞的函数。此函数的功能为允许调用者将自己的余额分配给其他用户。正常情况下, 第 5 行的断言能够保证调用者的余额足够支付待转出的总金额 *amount*。但是, 第 3 行对 *amount* 的计算过程中没有对数据溢出的情况进行检测, 从而导致可能发生溢出: 攻击者可以通过在调用时传递精心构造的参数, 让第 3 行的计算结果大于 EVM 的整数计算上限, 使得 *amount* 溢出为一个很小的值, 进而绕过第 5 行的检测, 并在第 7~9 行代码执行后获得巨量的数字资产。

2) 重入: 图 5 是一个存在重入漏洞的提款函数。此函数首先检查调用者是否有足够的余额, 然后通过 *call* 语句发送以太币, 并在最后扣除调用者相应的余额。如果有另一个合约 C 调用了该提款函数, 则 *call* 语句将触发合约 C 的回调函数, 并阻塞直到 C 的回调函数执行结束。此时调用者, 也就是 C 的余额并没有被扣除。如果 C 的回调函数再一次调用该提款函数, 则第二行的检测依然可以通过, 最终 C 可以取走原本不属于它的以太币。

```
function Cashout(uint _amount) public {
    require(balances[msg.sender] >= _amount);
    msg.sender.call.value(_amount)();
    balances[msg.sender] -= _amount;
}
```

图 5　重入漏洞样例

Figure 5　An example of reentrancy vulnerability

3) 不良随机源: 如图 6 所示的博彩合约中存在不良随机源漏洞。合约的作者在第 3 行使用

timestamp 作为随机源, 通过产生的随机数决定博彩游戏的胜负。然而在以太坊中很难找到合适的随机源。尽管无法准确地预测执行时的 *timestamp*, 但攻击者可以通过部署另一个执行相同计算的合约来测试第 3 行的条件是否会成立, 并在测试结果为成立时立即参与博彩, 从而立于不败之地。因此, 包含 *timestamp* 在内的区块上的数据都是不良的随机源, 使用它们产生随机数会导致安全风险。

```
1 function play() public payable {
2     require(msg.value >= 1 ether);
3     if (keccak256(timestamp) % 2 == 0) {
4         msg.sender.transfer(1.9 ether);
5     }
6 }
```

图 6　不良随机源漏洞样例

Figure 6　An example of bad randomness vulnerability

4) 访问控制: 正常情况下, 用于设置合约所有者的函数应当是特权函数或构造函数, 普通用户是无法调用这些函数的。但是, 在某些合约中, 这些函数却是每个人都可以访问的, 则导致每个人都可以成为合约的所有者, 进而可以执行某些特权操作。

5) 遗漏异常处理: 在某些情况下, 一个合约向另一个合约的调用可能会失败, 此时被调用合约中发生异常的信号会传递给调用者。这就要求调用方应当显式地检查返回值, 从而验证调用是否已成功执行并进行相应的处理, 否则可能导致意外的行为。

2.2　EVM 的数据处理

在智能合约字节码中, EVM 的指令只能对存储在某些位置的数据进行读取和写入, 这些位置包括栈(Stack), 内存(Memory)和外存(Storage)。

栈负责维护一系列大小为 32 字节的数据, 其最大深度为 1024。在 EVM 中, 大多数指令只能对存在于栈内的数据进行操作: 它们从栈中获取数据, 或是将结果存储到栈中, 例如 PUSH、ADD 等。

内存的作用类似于传统系统中的堆。通常, 内存用于处理超过 32 个字节的数据。例如, 指令 SHA3 可处理任意字节的数据, 此外还有一些指令(如 MLOAD 和 MSTORE)也能与内存进行交互。

与仅在执行时存在的栈和内存不同, 外存作为区块链的一部分, 与智能合约一同留存, 不会在合约执行结束后被销毁。外存维护合约的状态数据, 例如合约中用户的余额, 全局变量的值等。SSTORE 和 SLOAD 可用于读取与修改外存中的数据。

3　方案设计

DC-Hunter 的总体思路是从待测合约中提取模式, 并将其与已知漏洞合约中的模式进行匹配。但是, 由于智能合约的规模较小, 任何修改都会对匹配工作造成阻碍。例如, 图 4 中的漏洞函数包含 10 行代码, 而若想修复这个漏洞, 我们只需在第 3 行之后插入一条检查是否溢出的断言语句即可。修复后的函数与原版相比仅有 1 行存在差异, 二者显示出非常高的相似性。由于这种情况常常发生于真实的合约中, 因此我们利用切片技术来对函数模式进行提取, 从而减少噪声的影响, 增强匹配的效果。

图 7 是 DC-Hunter 的总体流程图。首先, 对待测合约和已知的漏洞合约进行预处理, 将合约字节码转化为指令集的形式, 并构建控制流图。随后, 遍历智能合约的 CFG, 找到并标记所有的特定指令作为切片条件; 通过对合约进行模拟执行, 确认数据与指令之间的关系, 以完成切片操作(3.1 节)。随后, 对所获得的切片进行规范化, 减少因使用不同版本编译器编译而导致的字节码差异(3.2 节)。训练一个图嵌入网络, 通过该网络得到所有规范化的切片的向量表示(3.3 节)。最后, 计算待测切片与漏洞切片两两之间的相似度。若一待测切片与某漏洞切片的相似度大于既定阈值, 则该切片所属合约会被输出为潜在的漏洞合约(3.4 节)。

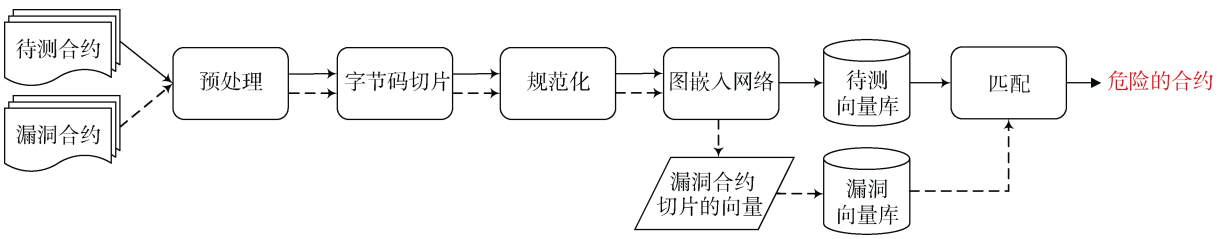


图 7　DC-Hunter 概述

Figure 7　An overview of DC-Hunter

3.1 合约切片

3.1.1 切片条件选择

任何来自合约外部的数据都有可能直接或间接的收到攻击者控制, 又或是很容易被攻击者知晓, 攻击者可以通过这些数据对智能合约进行攻击。考虑到这样的情况, 我们令所有从外部引入数据的指令都作为切片条件。具体的说, 切片条件可能引入以下 4 个种类的数据:

1) 交易数据。交易数据由调用方直接提供, 并通过 CALLDATALOAD、CALLVALUE 等指令引入执行。

2) 区块上的数据。在 EVM 中, 所有人都可以轻松获取区块上的数据。如果将它们用作随机源, 则攻击者可以通过模拟相同的随机数计算过程来破解随机性。引入区块上的数据的指令包括 BLOCKHASH、TIMESTAMP 等。

3) 外存数据。存储在外存中的数据通常在合约中扮演重要角色, 且可以被任何人访问, 例如用户的余额数据。指令 SLOAD 用于从外存中获取数据。

4) 外部调用返回值。外部调用的返回值应在调用者合约中进行显式地检查。执行外部调用的指令包括 CALL、SEND 等。

我们会先遍历一次智能合约, 标记所有在上述范围之内的指令, 作为后续进行切片的基础。

3.1.2 模拟执行并切片

在智能合约的字节码中, 绝大部分指令通过栈、内存和外存与数据进行交互, 除了 PUSH 指令外, 指令与数据的关联均没有在字节码中直接体现。然而, 进行切片需要分析指令与数据之间的依赖关系, 为了确认这样的依赖关系, 我们选择在模拟执行的基础上对智能合约进行切片。

对智能合约进行模拟执行, 具体地说就是模仿以太坊虚拟机的行为, 以命令的语义为基础, 在控制流图上依次执行待测合约中的指令; 模拟构造以太坊虚拟机的存储结构(栈、内存以及外存), 并在模拟执行的过程中同步地更新各个存储结构中的数据, 以辅助模拟执行期间各种指令功能的实现。在遇到需要引入外部数据的特定指令时, 用符号替代外部数据送入存储结构内; 在遇到条件分支时, 根据条件的情况来决定是仅执行某个分支或是分别执行两个分支。

在模拟执行智能合约的过程中, 即可进行切片工作: 对于由某个作为切片条件的指令 S, 其执行时引入的数据为 D, 在后续的执行过程中, 若有指令 I 将数据 D 或由 D 衍生出的数据作为参数, 则指令 I 纳入切片条件 S 的切片。

我们发现, 单纯的指令很多时候并不能体现其实现的功能, 例如 ADD 指令可能出现在加法计算、内存地址计算等场景中, 但单纯的 ADD 指令本身根本无法区分出这些场景。因此, 为了提升指令的区分度, 在形成切片时, 指令会额外地携带其参数信息。例如, 在前一段的例子中, 指令 I 会以“指令+参数”的形式纳入切片条件 S 的切片。

此外, 还存在一些关键的结构性指令, 如 REVERT、INVALID 等, 它们的出现预示着一次执行的结束, 往往对应着源代码中比较重要的行为, 因此我们也将它们考虑到了切片中。但这些指令不需要参数, 无法通过数据依赖关系将它们加入到切片中。因此, 对于每一个由切片条件引入的数据 D, 当一些结构性指令执行时, 若有数据 D 或由其衍生出的数据存在于任意存储结构中, 则将该结构性指令纳入数据 D 对应的切片。

3.2 规范化

在本节, 我们将对所获得的切片进行进一步的规范化操作, 从而减少因使用不同版本编译器编译而造成的字节码上的差异。

3.2.1 数值规范化

首先, 数据的具体数值将替换为代表数据属性的标签。这一步的目的是为了更好地描述数据的特征。在智能合约字节码中, 常数被广泛地使用, 例如 JUMP 指令的目标地址和内存地址的偏移量等。使用不同版本的编译器很可能会改变这些数据, 这将为我们的切片引入不必要的差异。此外, 仅凭具体数值难以描述数据的特点。而通过数据来自何处、经历过什么指令等信息, 我们可以使数据更具特征, 进而使指令更具区分性。

我们通过给数据打标签, 并让标签在执行过程中进行传播来实现数值的规范化。数据的标签取决于其所经历的指令, 表 1 展示了部分指令会赋予结果数据的标签。例如, TIMESTAMP 等指令引入的数据将被打上 blockdata 标签; 如果数据是由指令 GT 生成的, 则会被打上 compare_result 标签, 并同时继承作为 GT 参数的两个数据的标签。数据的所有标签即构成描述该数据的属性。该过程总共涉及 48 种指令以及 15 种标签。

3.2.2 参数顺序调整

在 Solidity 编译器版本升级时, 一些版本差异会新旧版本的编译器对相同的源码产生不同的字节码。例如, 在 Solidity v0.4.22[11]中有一项新功能: 如果比较运算符(LT、GT 等)的前面有 SWAP1 指令, 则

将该比较运算符替换为相反的运算符并移除 SWAP1 指令。

表 1 指令及对应的标签

Table 1 Instructions and corresponding tags

指令	赋予的标签
CALLDATALOAD, CALLDATACOPY	calldata
GASPRICE, BLOCKHASH, COINBASE, TIMESTAMP, DIFFICULTY, NUMBER, GASLIMIT	blockdata
CALLER, GAS, ORIGIN	transaction_data
ADD, MUL, SUB, EXP, SIGNEXT	compute_result
NOT, AND, OR, XOR, SHL, SHR, SAR	bit_operation_result
LT, GT, EQ, ISZERO	compare_result
MLOAD	from_memory
SLOAD	from_storage

为了减少这种不必要的差异, 我们首先令一个数据的所有标签按照字典序进行排序; 然后, 部分多参数指令的参数也将按其属性的字典序排序。在此过程中, 如果涉及到比较运算符的参数顺序更改, 则相应地将其替换为相反的运算符。具体涉及的指令可参见表 2。

表 2 参数顺序的重新排序

Table 2 Reordering of parameters

指令	调整方法
LT, GT, SLT, SGT	指令(attrB, attrA) → 相反指令(attrA, attrB)
ADD, MUL, EQ, OR, AND, XOR	指令(attrB, attrA) → 指令(attrA, attrB)

3.2.3 移除无关紧要的指令

在 EVM 中, 有两种类型的指令会直接对堆栈中的数据进行操作: DUP 类和 SWAP 类。由于编译器版本的差异, 这些指令很可能会在各个类内互相转化, 这种变化对匹配工作来书是不利的。

此外, 虽然 DUP、SWAP 和 POP 指令对栈内数据进行操作, 但它们与源代码行为的关联并不大。我们认为它们在代码匹配中的作用可以忽略不计, 因此表 3 所列出的指令将从切片中移除。

表 3 从切片中移除的指令

Table 3 Instructions to be removed from slices

类别	指令
DUP*x*	DUP1, DUP2, DUP3, DUP4, DUP5, DUP6, DUP7, DUP8, DUP9, DUP10, DUP11, DUP12, DUP13, DUP14, DUP15, DUP16
SWAP*x*	SWAP1, SWAP2, SWAP3, SWAP4, SWAP5, SWAP6, SWAP7, SWAP8, SWAP9, SWAP10, SWAP11, SWAP12, SWAP13, SWAP14, SWAP15, SWAP16
POP	POP

我们通过图 8 中的例子简要展示切片和规范化的效果。在该例中, 我们选择第一个指令 CALLDATALOAD 作为切片条件, 并通过分析哪些指令与其有数据依赖关系进行切片。进行规范化时, 切片中每个数据的具体数值被替换为属性标签; 指令 AND 和 LT 的参数顺序被调整, 同时 LT 被替换为 GT; 指令 DUP2 和 SWAP1 被从切片中移除。

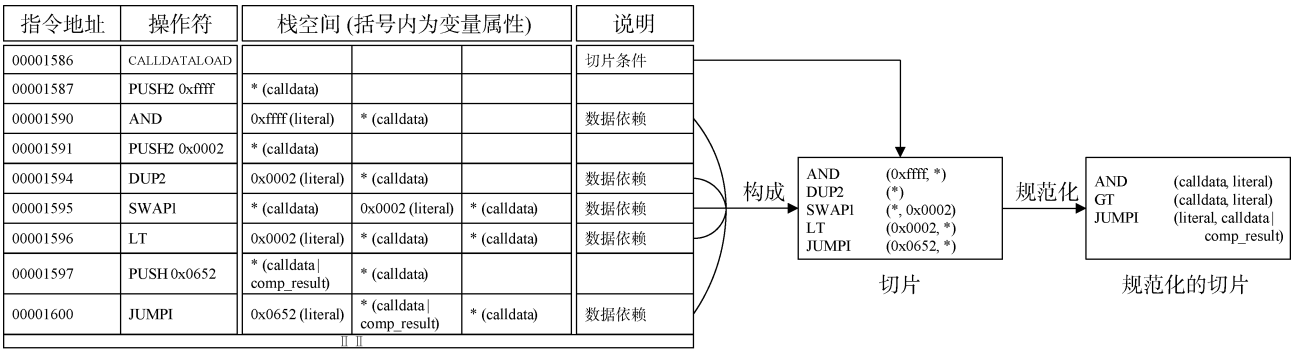


图 8 切片和规范化的示例

Figure 8 An example of slicing and normalization

3.3 图嵌入

经过切片和规范化操作后, 我们获得了较为纯净且形式统一的切片, 此时需要将切片嵌入向量空间, 以便于度量它们之间的相似度。在 Yamaguchi 等人[5,6]的工作中, 他们采用了 Bag-of-Feature 编码方法, 根据提取出的特征将函数映射为向量。这种编码方法假设所有的特征都是一个独立的维度, 导致特征之间的关系, 也就是函数的结构信息被忽略了。而在智能合约中, 许多指令有着较强的关联性, 而这些联系可以提供更为丰富的语义信息, 例如执行外部调用的指令与读写外存数据的指令有联系, 终止执行的指令通常受条件判断指令控制。

图 9 展示了图 3 与图 4 的函数中关键切片的结构。由于两个函数对总转账金额的计算方式不同, 它们切片中的指令存在明显差异, 若直接套用 Bag-of-Feature 方法, 则难以体现两个函数的相似性; 而考虑了结构信息之后, 指令之间的依赖关系进一步体现了二者源代码逻辑的相似, 从而提升这两个函数的相似程度。为了更好地捕捉切片内的结构信息, 我们将切片转化为程序依赖图的形式, 并在此之上通过图嵌入网络来计算切片的嵌入向量。

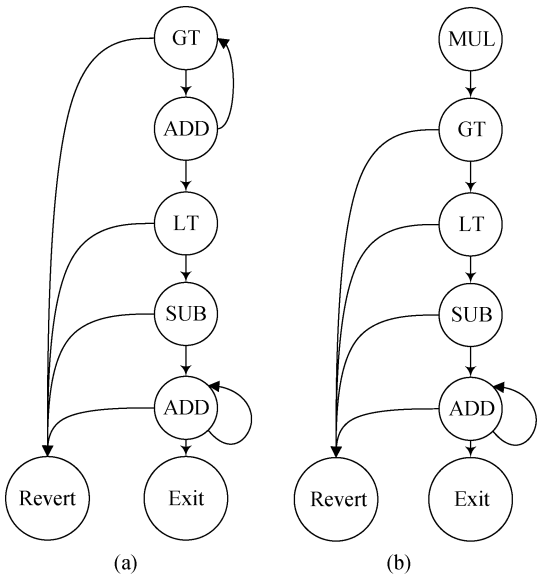


图 9　图 3 与图 4 中关键切片的结构

Figure 9　The structure of the key slices in Fig.3 and Fig.4

3.3.1　设计思路

为了学习切片的嵌入向量, 本文参考了已有的图嵌入方法 Graph2Vec[12]。Graph2Vec 是一种基于根子图(Rooted subgraph)抽取的无监督图嵌入算法, 可以有效地保留图的结构信息。对于一个图 G 中的某个节点 n 来说, 它的根子图就是节点 n 的所有邻居节点以及对应的边所构成的子图。与其他简单的低阶子结构(如节点)相比, 根子图提供了更为丰富的图结构信息。对于待嵌入的图 G, Graph2Vec 的目标是通过更新图 G 与根子图的嵌入向量, 使得 G 与其中的根子图有着尽可能相似的向量表示, 同时与 G 以外根子图的向量表示尽可能不相似, 从而实现对 G 的嵌入。

将 Graph2Vec 的思想应用到我们的工作中, 则前文中的图 G 就代表待嵌入切片的程序依赖图, 节点 n 代表切片中的指令, 根子图则表示一条指令及其上下文指令(以及它们之间的依赖关系)所构成的子图。训练的目标则是使得切片与其中根子图的向量的尽可能相似, 与该切片外其他根子图尽可能不相似, 从而得到切片的嵌入向量。

值得指出的是, 传统的 Graph2Vec 方法关注的是图的结构, 并不关注节点的属性值, 而在我们的图中, 作为节点的指令携带了丰富的信息。为了能够充分利用这些信息, 我们在计算根子图的向量时, 额外考虑了节点上的属性。具体地说, 我们将指令中的操作符与数据的标签直接映射为向量, 并将这些向量作为节点的属性, 参与根子图向量的生成。

3.3.2　算法概览

图嵌入网络的目标是将智能合约中所有的切片分别映射为向量。因此, 图嵌入网络的输入是包含待嵌入切片的智能合约 C, 以及嵌入向量的维度 S, 输出则是每个切片嵌入后所得到的 S 维向量。

图嵌入网络的流程为: 首先随机初始化所有在合约 C 中出现过的切片的嵌入向量, 并计算这些切片中所有根子图的嵌入向量; 然后进行两阶段的训练: 首先训练影响根子图嵌入的参数, 之后第二阶段将这些参数固定, 训练并输出切片的嵌入向量。

3.3.3　图嵌入网络设计

对于一个智能合约 $C = \{G_1, G_2, \cdots\}$, G_i 是从该合约中提取出的切片的程序依赖图, 定义为 $G_i = \{N_i, E_i, \lambda\}$, 其中 N_i 是切片中所包含指令的集合, E_i 是各个指令之间的依赖关系, λ 代表一个映射, 根据切片中指令的操作符和数据的属性, 将指令 n 映射为一个 S 维的向量 $\boldsymbol{v}_n$。图嵌入网络的流程如算法 1 所示。

算法 1　图嵌入网络

输入: 智能合约 $C = \{G_1, G_2, ...\}$
嵌入大小 S
输出: 每个切片 G 的嵌入向量 $\boldsymbol{v}_G \in R^S$

1. Randomly initialize $\boldsymbol{v}_G$ *FOR EACH* $G \in C$
2. FOR $G_i \in C$:
3. $SG_n = SubGraph(n, G_i)$ *FOR EACH* $n \in N_i$
4. FOR $G_i \in C$:
5. $J(\lambda, P) = -\log \sum_{n \in N_i} \Pr(SG_n \mid v_{G_i})$
6. $\lambda = \lambda - \alpha \cdot \partial J / \partial \lambda$
7. $P = P - \alpha \cdot \partial J / \partial P$
8. FOR $G_i \in C$:
9. $J(v_{G_i}) = -\log \sum_{n \in N_i} \Pr(SG_n \mid v_{G_i})$
10. $v_{G_i} = v_{G_i} - \alpha \cdot \partial J / \partial v_{G_i}$

首先, 随机地初始化所有切片的嵌入向量。算法 1 的第 2 至 3 行表示对于合约 C 的所有切片中每个指令节点的根子图, 计算它们的嵌入向量。其中, 第 3 行中根子图嵌入向量的计算方法如式(1)所示。

$$SubGraph(n,\ G_i) = \tanh(v_n + \boldsymbol{P} \sum_{m \in Neib(n,G_i)} v_m) \quad (1)$$

其中, $\boldsymbol{P}$ 是参数矩阵, $Neib(n,G_i)$是节点 n 在图 G_i 中的邻居节点。

接下来我们将通过两个阶段来获得切片的嵌入向量, 这体现在算法 1 中的第 5~10 行。该过程使用的概率函数如式(2)所示。

$$\Pr(SG_n \mid v_{G_i}) = \frac{\exp(v_{G_i} \cdot SG_n)}{\sum_{w \in N_C} \exp(v_{G_i} \cdot SG_w)} \quad (2)$$

其中, N_C 表示合约 C 中所有切片的节点的并集。

式(2)计算的是根子图 SG_n 出现在 G_i 切片中的概率, 训练的目标是使它最大化。如前所述, 我们将进行两个阶段的训练: 在第一个阶段中, 我们关注 λ 和 P, 仅对这二个参数进行训练; 在第二个阶段, 我们保持参数 λ 和 P 不变, 对切片的嵌入向量 $\boldsymbol{v}_G$ 进行训练。如此, 我们就获得了切片的嵌入向量。

3.4 危险合约检测

嵌入完成后, 合约的切片都转化为便于比较的向量形式。我们将所有待测合约的切片将构成待测向量库, 并从每个已知漏洞合约的切片中选出与漏洞关系最为密切的切片, 构成漏洞向量库。

之后, 我们通过余弦距离计算待测库与漏洞库中切片向量两两之间的相似度。此处得到的相似度值应属于[0, 1], 值越大, 则相应切片之间的相似度越高。若一待测切片与某漏洞切片的相似度大于阈值, 则该切片所对应的合约会被输出为潜在的漏洞合约。

4 实验评估

在本节中, 我们将使用部署在区块链上的真实智能合约来评估我们的方法。首先, 我们简要描述实验中的环境和使用的数据集(4.1 节)。其次, 我们通过实验证明 DC-Hunter 所使用的方法可以有效减少误报和误报(4.2 节)。最后, 我们展示了一个在 DC-Hunter 的帮助下找到的基于漏洞的蜜罐合约(4.3 节)。

4.1 实验环境及数据

我们在 C++环境中实现了合约预处理及切片, 在 Python 环境中实现了图嵌入网络以及危险合约检测。实验设备的配置为 4 GB 内存, 4 个 3.20GHz 内核和 2 TB 硬盘。

我们的实验基于以下 3 个数据集:

1) 数据集Ⅰ: 无源码合约。此数据集包含 200 余万个仅有字节码的智能合约。我们使用官方以太坊浏览器 Mist[13]来获取区块链数据, 并从中获取所有已部署在区块链上的合约字节码。

2) 数据集Ⅱ: 开源合约。虽然检测过程中我们仅使用合约的字节码, 但是开源合约中的源代码可大大提高结果审计的准确性。一些合约作者在以太坊浏览器 Etherscan 上验证并公开了其合约的源代码。该数据集中的合约是在 Etherscan 政策调整之前收集的。由于有些合约被多次验证并公开, 我们从数据集中删除了重复的合约。最终, 该数据集包含 32,499 个开源的智能合约。

3) 数据集Ⅲ: 漏洞合约。此数据集中的合约将用作匹配的种子。我们主要通过检索 CVE 来获得大量的漏洞合约, 并从中筛选出了一部分会造成真实危害, 且具有代表性的合约。最终, 该数据集共包含 24 个合约, 其中 10 个涉及整数溢出漏洞, 4 个涉及重入漏洞, 6 个涉及不良随机源漏洞, 2 个涉及访问控制漏洞, 2 个涉及遗漏异常处理漏洞。为了使实验结果更加合理, 我们已从前两个数据集中剔除了在数据集Ⅲ中出现的所有合约。

4.2 实验结果

4.2.1 实验效率评估

首先, 我们对数据集 I 进行了切片和规范化, 此过程耗时约 57 个小时。图 10(a)展示了切片长度的累积分布, 图 10(b)展示了部分指令使用次数的分布。

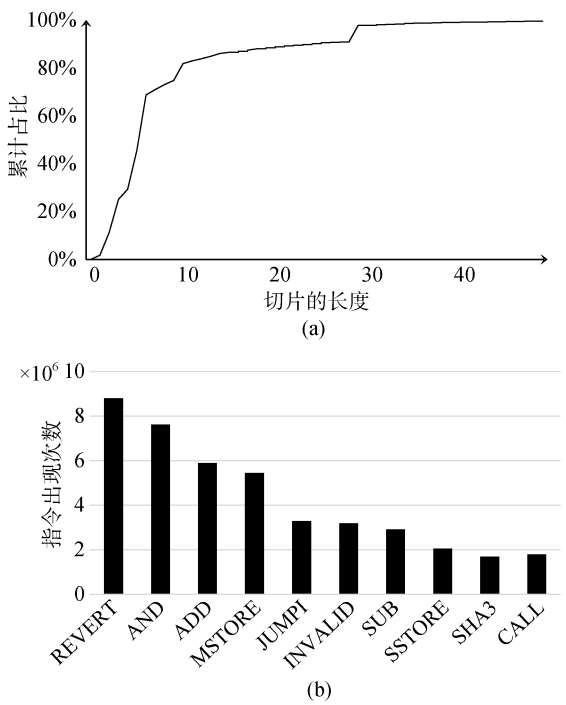


图 10 切片长度与指令使用次数统计

Figure 10 Statistics about the length of slices and the usage time of instructions

可以发现, 大多数切片的长度均小于 10, 并且许多指令出现了多次, 这两个现象表明切片中单纯的指令不具有足够的代表性, 因此规范化的过程中, 我们通过带有属性标签的参数来区分指令。

我们也对数据集Ⅱ中合约的字节码进行了切片和规范化操作, 花费的时间约为 35 分钟, 所获得的切片与数据集Ⅰ有着相似的统计分布。之后, 我们通过图嵌入网络将规范化的切片嵌入向量空间中, 花费时间约为 200 分钟。

我们进一步对数据集Ⅲ进行切片, 筛选出代表漏洞的切片作为匹配的种子, 从而建立漏洞向量库, 然后与数据集Ⅰ的切片向量之间进行相似度计算。每一个已知漏洞合约的所花费的检测时间平均为约 17 分钟。

4.2.2 实验结果分析与对比

我们首先从数据集Ⅱ中随机选择了 3,000 个合约, 使用 DC-Hunter 进行漏洞检测, 并计算不同阈值情况下的误报率与漏报率。通过调整这二个指标之间的平衡, 相似度阈值最终被设定为 90%。此外, 为了平衡效率和效果, 我们还对图嵌入网络中的嵌入大小 S 进行了调参, 最终 S 的取值为 64。这些参数在日后的使用中都可以根据需求进行调整, 此处的设置则可作为有价值的参考。

对于数据集Ⅰ, DC-Hunter 共报出 1,220 个漏洞合约, 其中整数溢出、重入、不良随机源、访问控制和遗漏异常处理漏洞的数量分别为 732、129、23、182 和 154 个。为了审计这些无源码合约, 我们将其部署在私链上, 并尝试对其中存在的漏洞进行利用。实际上, 手动检查底层字节码是非常耗时且不可靠的, 因为这要求我们能够通过字节码推测源代码中的行为, 推断漏洞的位置以及触发方法, 然后才能对漏洞的存在进行测试。因此, 我们仅手动检查了排名靠前的 10 个与已知漏洞合约相似度最高的合约, 并成功触发了 8 个合约中的漏洞。

对于数据集Ⅱ, 我们总共发现 152 个可利用的漏洞合约, 对应 5 种漏洞的数量分别为 104、23、10、11 和 4 个。结果中还包含 19 个误报, 因此本方法应用于数据集Ⅱ的准确率为 88.89%。

为了证明 DC-Hunter 所采取策略的有效性, 我们基于 Bag-of-Feature 方法进行了两个比较实验, 并与广泛应用的工作 Oyente 进行比较, 实验在数据集Ⅱ上进行评估。下面介绍实验的过程和相应的结果。

对比实验: 切片。在该对比试验中, 我们移除了 DC-Hunter 中的切片步骤。也就是说, 每个合约都在转换为指令后, 直接进行规范化操作, 随后使用合约的整个指令集进行相似性比较。结果如下:

表 4 是在不同切片设置下二个实验的表现情况, 其中每组数字的右侧为报出的潜在漏洞合约数量, 左侧为审计后确认的漏洞合约数量。可以看到, DC-Hunter 在各种漏洞上的性能均优于无切片的方法, 尤其是大幅削减了误报的产生, 这说明了切片技术的有效性。

表 4 不同切片设置的实验结果

Table 4 Result of different slicing settings

	整数溢出	重入	不良随机源	访问控制	未检查返回值
DC-Hunter	104/112	23/26	10/16	11/13	4/4
无切片实验	97/479	14/79	9/53	6/32	2/31

对比实验: 规范化。在本组对比实验中, 我们从 DC-Hunter 中移除了规范化步骤, 但保留了切片步骤。

表 5 的实验结果显示, DC-Hunter 在各种漏洞上的表现都优于无规范化的版本, 大幅提高了检测出漏洞的数量。为了更好地体现实验方法的效果, 我们额外进行了相似度阈值设置为 100%时的实验。在这组实验中, DC-Hunter 依然可以捕获到比未规范化时更多的漏洞合约, 这表明存在这么一部分合约, 它们在规范化之前是不相同的, 而规范化操作合理地消除了它们之间的差异。

表 5 不同规范化设置的实验结果

Table 5 Result of different normalization settings

	整数溢出	重入	不良随机源	访问控制	未检查返回值
DC-Hunter (阈值 100%)	86/86	14/14	3/3	11/11	3/3
无规范化实验 (阈值 100%)	10/10	3/3	3/3	9/9	3/3
DC-Hunter (阈值 90%)	104/112	23/26	10/16	11/13	4/4
无规范化实验 (阈值 90%)	76/84	17/20	3/7	11/13	3/3

对比实验: Oyente。我们与目前应用最广泛的分析工具之一: Oyente[14]进行了比较。由于只提供字节码时, Oyente 仅能够检测第 2 章所提及 5 种漏洞类型中的重入漏洞, 因此我们只针对重入漏洞的检测进行比较。对于数据集Ⅱ中的字节码, Oyente 共报出了 172 个潜在漏洞合约。通过手动审计, 我们认为其中有 18 份合约是真正的漏洞合约, 而剩余的则是误报, 因为这些合约中的漏洞实际上是无法利用的。

4.2.3 案例研究: BattleToken 合约

图 11 为 DC-Hunter 使用图 4 所示的漏洞合约作为种子时新发现的漏洞合约, 名为 BattleToken, 与种子合约的相似度为 97.6%。BattleToken 合约是部署于以太坊上的一个游戏合约, 其中的 *batchTransfer* () 函数允许调用者将他的余额分配给多个人。在此函数中, 开发人员在第 2 行使用 *safeSub* ()函数来对调用者的余额进行扣除——当调用者余额不足时, 该次调用会被回滚。

```
1  function batchTransfer(address[] _to, uint _value) public {
2      balances[msg.sender] = safeSub(
3          balances[msg.sender], _to.length * _value);
4      for (uint i = 0; i < _to.length; i++) {
5          balances[_to[i]] = safeAdd(balances[_to[i]], _value);
6          Transfer(msg.sender, _to[i], _value);
7      }
8  }
9  function safeSub(uint256 a, uint256 b) internal returns
       (uint256) {
10     assert(b <= a);
11     return a - b;
12 }
```

图 11 BattleToken 合约[15]

Figure 11 Contract BattleToken[15]

然而, 为了计算调用者需要支付的总金额, 开发者在第 3 行使用了一个直接的乘法, 并且没有进行任何的溢出检查。显然, 这样的乘法会导致整数溢出漏洞。通过构造输入对漏洞进行利用, 攻击者能够将任意用户的余额设置为任何值。

可以看出, 图 11 与图 4 中二个合约的源码并不相似; 实际上, 在字节码层面它们也并不相似: 通过直接比较指令集, 它们的相似度仅有 71%。但是, 由于参数 *_value* 在这两个函数中经历的处理非常相似, 因此借助切片等技术, DC-Hunter 可以找到该漏洞合约。此漏洞已得到其开发人员和 CVE 的确认。

4.2.4 案例研究: 一个闭源合约

在本节中, 我们将通过一个例子来简要说明手动审计纯字节码合约的过程。

DC-Hunter 将闭源合约[16]标记为潜在的漏洞合约。为了验证其正确性, 首先我们需要确认漏洞函数的调用接口。幸运的是, 通过将函数入口点的指纹与其他合约中的指纹进行比较, 我们得知其调用接口为 *batchTransfer* (*address*[], *uint256*)。之后需要确认漏洞的利用方式。在该合约中, 存在漏洞的切片与 BecToken 的漏洞切片非常相似, 因此我们以相同的方式进行尝试。最后, 我们需要确认攻击利用是否成功。通过比较入口点指纹, 我们发现该合约拥有 *balanceOf* (*address*)的调用接口, 我们可以使用它来查看各个账户的余额, 从而验证先前的操作确实已经导致了数据溢出, 最终确认了漏洞的存在。

然而, 在绝大多数情况下, 上述的步骤很难如此顺利, 这会极大地增加手动审计纯字节码的难度。因此, 在 4.2.2 节中, 我们将评估的重点集中在了数据集 II 上。

4.3 一个蜜罐合约

在实验过程中, 我们发现一些合约虽然被标记为潜在的漏洞合约, 但是经过验证, 这些合约的漏洞并不能被成功利用。此外, 这些合约还会使得攻击者在尝试进行攻击时遭受损失。我们将这些合约称为蜜罐合约。

例如, 图 12 是一个以太坊游戏合约的摘要。想要成为该游戏的参与者, 用户需要支付不低于 *price* 数量的以太币; 此外, 参与者可以再次支付同样数量的以太币, 以获得奖励并退出游戏。参与游戏持续的时间越长, 退出时的奖励越高。

```
1  mapping (string => uint) parameters;
2  function set_parameter(string name, uint value) public{
3      require(msg.sender == address(parameters['owner']));
4      parameters[name] = value;
5  }
6
7  function claim_reward(uint uid, bytes32 passcode) public
       payable
8  {
9      require(msg.value >= parameters["price"]);
10     require(is_passcode_correct(uid, passcode));
11     uint final_reward = get_reward(uid) + msg.value;
12     if (final_reward > parameters["price_pool"])
13         final_reward = parameters["price_pool"];
14     require(msg.sender.call.value(final_reward)());
15     /* delete the user */
16 }
```

图 12 HODLerParadise 合约[17]

Figure 12 Contract HODLerParadise[17]

在此合约中, 开发人员使用映射 *parameters* 来存储合约中的关键参数。开发人员可以通过第 2 行的 *set_parameter* ()函数来对此映射中的值进行设置, 此外任何参与者都可以查看到该映射中的值。当参与者想要退出并赢取奖金时, 他应调用第 7 行的函数 *claim_reward* ()。此函数的逻辑为, 首先进行一些检查, 然后计算参与者应得的奖金, 并保证奖金金额不大于奖池的余额, 随后将相应的以太币发送给参与者, 最后删除参与者的数据。

由于此合约使用 *call* ()发送以太币(第 14 行), 并在发送之后才删除用户帐户(第 15 行), 且没有任何防御机制, 因此该合约存在重入漏洞。

然而, 实际上这是一个蜜罐合约。重点在于第 12 行的条件判断语句, 它的作用似乎是当待提取的奖金大于奖池余额时, 调整奖金的数量。但是, 在第 12、13 行中, 单词“price_pool”的第二个字符‘o’并

不是英文字符, 而是读作“omicron”的希腊字符, 这二个字符在以太坊浏览器 Etherscan 的代码栏中, 外形完全相同, 但实际上此“price_pool”非彼“price_pool”。这个伪装成奖池余额变量的值永远是 0, 于是第 12、13 行代码就使得待提取的金额一定会被设置为 0, 从而所有对该函数的调用都无法提取出任何奖金。最终, 所有攻击者为了利用漏洞而支付的以太币将被锁定, 并且会被合约的所有者通过后门函数取走。

5 讨论

漏洞类型。虽然在本工作中我们仅检测了 5 种类型的漏洞, 但实际上 DC-Hunter 可检测的漏洞类型包括所有因对外部数据的不恰当操作而导致的漏洞。然而, 仍有某些漏洞类型不在我们的检测范围内, 例如在 Maian[18]工作中首次提及的“贪婪合约”。贪婪合约的特征是可以接收以太币, 但在整个合约中均无法释放以太币。这种漏洞的特征无法通过单个切片来描述, 因此不在我们的检测范围内。我们可以在匹配过程中使用更多维度的信息, 从而扩展可检测的漏洞范围。

误报。在审计过程中, 我们发现有些误报与已知漏洞切片很相似, 但实际上它们的功能完全不同。例如, 在不良随机源的例子中, 有一些误报的产生是因为在字节码层面, 计算随机数与写入外存数据这二种操作的指令序列十分相似, 从而会产生混淆。为了减少这种误报, 我们可以对数据的属性类型进行更加细致的划分, 从而使得不同操作的切片之间更有区分度。

蜜罐合约的验证。在本工作中, 我们验证蜜罐合约的方法为, 将潜在的漏洞合约部署至私有链上, 并提供类似于主链上的环境, 从而检验漏洞是否可利用。我们可以通过一些自动化的方法, 从合约是否会锁定用户以太币等角度进行检测, 从而更加可靠而高效地寻找蜜罐合约。

6 相关工作

如前文所述, 匹配技术在程序分析中非常有效, 并已被广泛用于检测漏洞等方面[19-21]。在源代码层面, Yamaguchi 等人[5-6]里程碑式的工作中提出了“漏洞外推”概念, 在这些工作中, API Token 和抽象语法树被提取为函数的特征, 随后特征将被编码为向量, 并计算特征之间的相似度。在字节码层面, 许多工作都是基于特征提取和图匹配进行的。discovRE[22]直接对程序控制图的结构进行比较。与之相对的是, Genius[4]并非直接进行图匹配, 而是提取每个基本块的统计数据, 以此形成属性控制流图, 并通过谱聚类生成码本, 将图编码为向量后才进行比较。由于缺乏源代码和噪声等问题, 这些检测方法无法直接应用于智能合约领域。值得一提的是, Genimi[23]使用了 Structure2Vec 方法与 Siamese 架构以构建图嵌入网络, 从而将函数映射为向量并进行克隆检测。该方法可实现跨平台检测的效果, 这与智能合约领域中的编译器版本问题相对应。然而, Genimi 仅使用 CFG 中基本块的统计数据作为节点的属性, 并基于这些属性将 CFG 映射为向量。我们认为使用统计数据代表基本块的做法会导致信息的丢失, 因此在将指令转化为向量时, 我们采用了直接映射的方式。

在智能合约领域, 此前关于漏洞检测的工作可以通过其基本方法进行分类, 其中符号执行是一种比较常见的方法[12,24,25]。基于符号执行的工作可能会有着更高的准确性, 但是由于符号执行的特性, 不可避免地要花费大量时间。例如, Oyente[12]需要花费大约 3,000 个小时来分析 19,366 个智能合约, 而我们可在十余个小时内分析上百万个智能合约。此外, 还有一些基于交互式定理证明的工作[26-29], 这些工作各有优势, 但需要大量的人工干预, 因此也不适用于大规模的智能合约分析工作。基于抽象解释的方法[30]不需要人工干预, 但可能会引入较多的误报。在我们的工作中, 仅在选择漏洞合约的关键切片时需要少量的人工干预, 这种程度的人工干预不会影响工作的可伸缩性, 并且可以大幅削减噪声的影响、降低误报。

近期还有一些关注蜜罐合约的工作。Sanjuas[31]列出了他遇到的数个蜜罐合约, 并解释了其中的陷阱所在。Torres 等人[32]总结了几种蜜罐合约, 并基于抽象特征构建了一个检测工具。在我们的工作中, DC-Hunter 可以在没有事先总结的特征的帮助下, 系统地对蜜罐合约进行检测。

7 结论

区块链上危险的智能合约可能会导致用户的资产损失, 因此检测智能合约漏洞是一项至关重要的任务。由于缺乏完善的漏洞特征库, 匹配作为一种不需要事先总结漏洞特征的方法, 成为良好的解决方案。然而, 智能合约领域的字节码匹配将面临以下问题: 匹配过程更容易受到噪声的影响; 版本繁多的编译器被用于智能合约的编译, 而这会导致生成的字节码存在差异; 忽略函数的结构特征会导致难以捕获到某些漏洞。在本文中, 我们提出了 DC-Hunter,

一种基于字节码匹配的智能合约漏洞检测方案，并实现了它的原型。DC-Hunter 使用切片技术来大幅降低噪声的影响，并对生成的切片进行规范化，从而减少编译器版本不同所造成的差异，此外还使用图嵌入技术捕捉代码之间的结构相似性，进一步提升了漏洞检测的能力。我们使用真实的智能合约对 DC-Hunter 进行了评估，发现了大量的漏洞合约，以及许多蜜罐合约。

参考文献

[1] The Biggest Crowdfunding Project Ever—the DAO—Is Kind of a Mess. https://www.wired.com/2016/06/biggest-crowdfunding-project-ever-dao-mess, Apr 2020.

[2] OVERFLOW ERROR SHUTS DOWN TOKEN TRADING. http://hiphopworldmagazine.com/homeposts/overflow-error-shuts-down-token-trading, Apr 2020.

[3] Etherscan-Verified contract list. https://etherscan.io/ contractsVerified, Apr 2020.

[4] Feng Q, Zhou R D, Xu C C, et al. Scalable Graph-based Bug Search for Firmware Images[C]. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 2016: 480-491.

[5] Yamaguchi F, Lindner F, Rieck K, Vulnerabili-ty extrapolation: assisted discovery of vulnerabilities using machine learning[C]. *The 5th USENIX conference on Offensive technologies, USENIX Association*, 2011:456-462.

[6] Yamaguchi F, Lottmann M, Rieck K. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees[C]. *Proceedings of the 28th Annual Computer Security Applications Conference*, 2012: 3-7.

[7] Beercoin. https://etherscan.io/address/0x7367a68039d4704f30bfbf6d948020c3b07dfc59, Apr 2020.

[8] Remix the Solidity IDE. http://remix.ethereum.org, Apr 2020.

[9] Ammbr. https://etherscan.io/address/0x96c833e43488c986676e9f6b3b8781812629bbb5code, Apr 2020.

[10] BecToken. https://etherscan.io/address/0xc5d105e6 3711398af9bbff092d4b6769c82f793d, Apr 2020.

[11] Solidity change log. https://github.com/ethereum/solidity/blob/develop/Changelog.md, Apr 2020.

[12] Narayanan A, Chandramohan M, Venkatesan R, et al, graph2vec: Learning distributed representations of graphs, arXiv preprint arXiv:1707.05005, 2017.

[13] Mist. https://github.com/ethereum/mist/, Apr 2020.

[14] Luu L, Chu D H, Olickel H, et al. Making Smart Contracts Smarter[C]. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, 2016: 254-269.

[15] BattleToken. https://etherscan.io/address/0x4daa9dc438a77bd59e8a43c6d46cbfe84cd04255, Apr 2020.

[16] A close-source contract. https://etherscan.io/address/0xddc36be766d4328dbf898cfd9826478659dba5c3, Apr 2020.

[17] HODLerParadise. https://etherscan.io/address/0xc03b0dbd201ee426d907e367f996706cf53b8028, Apr 2020.

[18] Nikolić I, Kolluri A, Sergey I, et al. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale[C]. *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018: 653-663.

[19] Jang J, Agrawal A, Brumley D. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions[C]. *2012 IEEE Symposium on Security and Privacy*, 2012: 256-262.

[20] Kim S, Woo S, Lee H, et al. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery[C]. *2017 IEEE Symposium on Security and Privacy (SP)*, 2017: 595-614.

[21] Li Z, Lu S, Myagmar S, et al. CP-Miner: Finding Copy-paste and Related Bugs in Large-scale Software Code[J]. *IEEE Transactions on Software Engineering*, 2006, 32(3): 176-192.

[22] Eschweiler S, Yakdan K, Gerhards-Padilla E. DiscovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code[C]. *Proceedings 2016 Network and Distributed System Security Symposium*, 2016: 256-261.

[23] Xu X J, Liu C, Feng Q, et al. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection[C]. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017: 363-376.

[24] Grishchenko I, Maffei M, Schneidewind C. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts[M]. Computer Aided Verification. Cham: Springer International Publishing, 2018: 51-78.

[25] Kalra S, Goel S, Dhawan M, et al. ZEUS: Analyzing Safety of Smart Contracts[C]. *Proceedings 2018 Network and Distributed System Security Symposium*, 2018: 230-236.

[26] Amani S, Bégel M, Bortin M, et al. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL[C]. *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018: 66-77.

[27] Grishchenko I, Maffei M, Schneidewind C. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts[M]. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018: 243-269.

[28] Hildenbrandt E, Saxena M, Zhu X, et al, Kevm: A complete semantics of the ethereum virtual machine. Technical Report, 2017.

[29] Hirai Y. Defining the Ethereum Virtual Machine for Interactive Theorem Provers[M]. Financial Cryptography and Data Security. Cham: Springer International Publishing, 2017: 520-535.

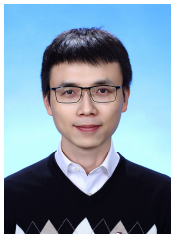
[30] Grech N, Kong M, Jurisevic A, et al. MadMax: Surviving Out-of-gas Conditions in Ethereum Smart Contracts[J]. *Proceedings of the ACM on Programming Languages*, 2018, 2(OOPSLA): 1-27.

[31] An analysis of a couple Ethereum honeypot contracts. https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d, Apr 2020.

[32] Torres C F, Steichen M. The Art of The Scam: Demystifying Honeypots in Ethereum Smart Contracts. arXiv preprint arXiv: 1902.06976, 2019.

韩松明 于 2017 年在中国人民大学信息安全专业获得学士学位。现在中国人民大学信息安全专业硕士学位。研究领域为软件安全性分析。研究兴趣包括: 软件安全、区块链安全等。Email: songming_h@qq.com

梁彬 于 2004 年在中国科学院软件研究所计算机软件与理论专业获得博士学位。现任中国人民大学信息学院教授。研究领域为信息安全。研究兴趣包括: 软件安全性分析、系统软件安全及人工智能安全性等。Email: liangb@ruc.edu.cn



黄建军 于 2017 年在普渡大学计算机科学专业获得博士学位, 现为中国人民大学信息学院计算机系讲师, 研究兴趣包括: 移动安全、区块链安全、软件安全分析。Email: hjj@ruc.edu.cn

石文昌 于 2002 年在中国科学院软件研究所计算机软件与理论专业获得博士学位。现任中国人民大学信息学院教授。研究领域为信息安全。研究兴趣包括: 信息安全、可信计算和数字取证等。Email: wenchang@ruc.edu.cn