计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330613
61(2): 274−288, 2024

DEALS——追踪代币转账信息不一致

姜人楷[1] 宋书玮[1] 罗夏朴[2] 陈 厅[1] 罗瑞杰[1] 王炳森[1] 乔 翱[1]
[1](电子科技大学网络空间安全研究院 成都 611731)
[2](香港理工大学电子计算学系 香港特别行政区 999077)
(202121080527@std.uestc.edu.cn)

DEALS——Track Token Transfer Information Inconsistency

Jiang Renkai[1], Song Shuwei[1], Luo Xiapu[2], Chen Ting[1], Luo Ruijie[1], Wang Bingsen[1], and Qiao Ao[1]
[1](*Cyberspace Security Research Institute*, *University of Electronic Science and Technology of China*, *Chengdu* 611731)
[2](*Department of Computing*, *the Hong Kong Polytechnic University*, *Hong Kong SAR* 999077)

Abstract Blockchain enables traditional exchanges and lending houses to be extended to Depl (decentralized platforms), which allows anyone to access exchange and lending without the help of intermediaries. Most Depls are implemented as smart contracts running on Ethereum and interact with another smart contract, cryptocurrency (i.e. token), to achieve various functions. Although Depl involves more than 35 billion worth of tokens, little is known about whether the actual transfer of tokens is as consistent as Depl expects. The inconsistency between the actual transfer of tokens and what the decentralized platform expects is known as behavioral inconsistency, resulting in property damage and user confusion. In this work, we take the first step to investigate such inconsistency between Depl and tokens. We propose to automatically detect inconsistency by comparing the actual token transfer behavior with the behavior indicated by the internal records of Depl by monitoring the core data structure changes of Depl and token.The experimental results show that inconsistent behavior exists in 1 012 749 transactions with an accuracy of 98.0%, which involves 2 871 pairs of Depl and tokens, and is related to 110 Depl and 2 544 tokens.In addition, 10 main reasons behind the inconsistency are summarized, e.g., cheater Depl, inconsistent scale, unclear coin lock rules, etc.

Key words block chain; decentralized platform (Depl); token; inconsistent behavior; automatic detection

摘 要 区块链使传统的交易所和借贷机构能够扩展到去中心化平台(decentralized platform, Depl),任何人都可以在没有中介机构帮助的情况下进行交易和借贷.大多数Depl都是作为运行在以太坊上的智能合约实现的,并与另一种智能合约、加密货币(即代币)交互,以实现各种功能.尽管Depl涉及价值超过350亿的美元,但人们对代币的实际转移是否如Depl预期的那样一致知之甚少.代币的实际转移与Depl期望的不一致被称为行为不一致,这种不一致的出现将导致财产的损失和用户的质疑.在这项工作中,我们迈出了调查Depl和代币之间的这种不一致的第一步.我们提出通过监控Depl和代币的核心数据结构变化,将实际的代币转移行为与Depl内部记录指示的行为进行比较,自动检测不一致.实验结果表明,在1 012 749笔交易中存在不一致行为,涉及2 871对Depl和代币,与110个Depl和2 544个代币相关.实验结果的精准度为98.0%.此外,还总结了导致不一致的10大原因,例如诈骗Depl、Depl与代币尺度不一致、锁币规则不明确等.

关键词 区块链;去中心化平台;代币;行为不一致;自动检测

中图法分类号 TP391

收稿日期: 2023-07-26; **修回日期**: 2023-09-21

通信作者: 陈厅(brokendragon@uestc.edu.cn)

由于区块链的去中心化、开放性和匿名性，传统交易所和借贷机构正在扩展到去中心化平台（decentralized platform, Depl）. 为传统交易所和借贷机构提供了新功能，例如允许任何人无需中介即可获得服务[1]. 由于 Depl 拥有着大量的投资者，去中心化生态中有 70% 的价值是被锁定在 Depl 中的，这些价值总计超过 355.7 亿美元（其中去中心化交易所占 215.1 亿美元，去中心化借贷平台占 140.6 亿美元[2]）. 而根据知名去中心化金融数据平台 Defillama 的数据显示，大多数的 Depl 是运行在以太坊区块链上的：资金排名前 50 名的去中心化交易所中有 12 个运行在以太坊上，前 50 名去中心化借贷平台中有 18 个运行在以太坊上[2]. 这些 Depl 都是基于运行在以太坊上的智能合约实现的[2]. 智能合约是在以太坊区块链上运行的程序[3]. 传统平台一旦实现为智能合约，就可以根据预先编写的程序提供预期的服务.

本文的工作重点是去中心化交易所和去中心化借贷平台，我们将它们统称为 Depl，它们实际上是中心化平台在以太坊区块链上的智能合约.

Depl 提供多种功能，包括资产交换和抵押资产进行借贷，其与传统平台类似. 在区块链背景下，资产以代币（token）的形式实现，代币本质上是具有所有权账本、使用账本记录资产转移信息的智能合约[4-5]. 因此，用户之间转移代币的过程实际上是代币合约接口被用户调用后，在代币账本中记录资产转移的过程，该过程简称为代币转移. 为了规范化该过程，以太坊改进提议（EIP）已经提出了多个标准来指导此类代币接口的实现. Depl 功能的实现依赖于 Depl 与代币接口进行交互，进而使代币账本更新用户资产，将用户资产所有权在 Depl 和其他用户之间周转，最终实现金融功能. 调查发现，2020—2022 年之间最流行的前 60 个 Depl[6] 都通过 EIP 定义的标准接口与代币合约进行交互.

然而，虽然代币标准为 Depl 和代币之间的交互提供了便捷的方式，但它并不一定保证代币的实际转移符合 Depl 的预期. 本文分析了 60 个 Depl 的白皮书、文档（或网页），没有发现任何 Depl 描述此问题. 对于 Depls 和 Depls 的用户来说，他们会直接默认一致性.

不幸的是，事实证明，确实存在一些代币实际转移与 Depl 的预期不符的事件. 发生此类事件的后果可能将导致代币非法转让，使用户和 Depl 的财产遭受损失. 例如，恶意行为者利用名为 HYDRO 的有缺陷的代币进行虚假存款，而该代币由于合约中的错误而未转移. 然而，名为 Etherdelta 的 Depl 认为，由于代币成功调用了代币的转账接口，存入了超过 26 亿枚代币. Etherdelta 在自己的合约数据结构中记录了错误的代币转账信息，直到 2019 年才知道有错误.

我们分析了与 Depl 相关的攻击案例，发现在一些攻击案例中，Depl 惯于使用自己合约内部的数据结构来记录代币转移信息. 这和代币的所有权账本非常相似. 我们在第 4 节中对 2020—2022 年最流行的 Depl 进行了手动分析，发现 85% 的 Depl 使用这种方法记录代币转移信息. 就代币而言，EIP 规定了代币使用统一的数据结构来记录代币转移信息，但是 Depl 没有相关的规定.

因此，受案例和人工分析的启发，本文通过研究 Depl 合约中记录代币转移信息的数据结构、该数据结构改变量和代币合约中的数据结构改变量之间的关系来探索未发现的安全问题或财产风险. 对于正式的描述，“核心数据结构”用于描述“记录代币转移信息的数据结构”. “行为”被定义为“智能合约对合约自身的核心数据结构进行修改”. 本文探讨了 Depl 与代币行为的不一致，即 Depl 预期代币转移与代币本身的转移之间的差异.

本文提出了一种自动化方法，通过实现名为 DEALS（dex and lending scope）的工具来检测 Depl 和代币之间的行为不一致. 实现自动检测不一致存在 2 个挑战：第 1 个挑战是自动识别 Depl 应用程序的核心数据结构. 由于 Depl 的实现没有标准化的协议，因此其核心数据结构的相关信息未知. 这就导致了定位 Depl 核心数据结构困难. 第 2 个挑战是在多合约交互过程中将 Depl 和代币的行为分别关联起来. 由于合约交互信息只能在运行时获取，现有的静态分析方法很难识别合约交互的行为.

为了解决第 1 个挑战，该工作从排名靠前的 Depl 文档和开源代码中总结了为 Depl 提供核心功能的函数，即核心函数. 同时，该工作总结了 Depl 常用于存储代币转账信息的数据结构类型，即核心数据结构. 通过利用 Depl 的核心功能和核心数据结构模式对区块链上的智能合约进行过滤，可以达到自动识别核心数据结构的目的.

对于第 2 个挑战，通过使用完整节点来重放交易并研究合约交互以跟踪 Depl 和代币的行为来解决. 该方案定义适用于 Depl 的行为关联规则，然后检测每条追踪路径的不一致行为.

本文的主要贡献包括 4 个方面：

1）据我们所知，本文工作是第 1 个检测 Depl 和

代币之间不一致行为的工作. 研究了 Depl 的核心功能和核心数据结构，发现了造成财产损失或导致用户迷惑的 10 个不一致的原因.

2) 通过分析得到了 Depl 的 3 种主流核心数据结构，发现属于这 3 种核心数据结构的不同的 Depl 有 3 957 个.

3) 通过使用 DEALS，对区块高度在 5 万~1 200 万之间的所有交易进行了检测. 实验结果表明，有 1 012 749 笔交易存在不一致行为，这关联到 2 871 组 Depl 和代币. 通过手动分析所有触发不一致的对，只有 57 对误报，因此 DEALS 的精准率高达 98.0%.

4) 总结了不一致的行为，揭示了 10 个主要原因，包括尺度不一致、锁币规则不明确等.

1 相关工作

在本文工作之前，已经有一些针对智能合约安全检测的工作. 一些安全分析工具被用于检查代码漏洞(例如，重入性、整数溢出、未经检查的调用等)，比如 TokenScope [7]，oyente[8]，Manticore[9]，VeriSmart[10] 和 MythX[11]；一些工具专注于判断合约行为的合法性，比如 Sereum[12] 和 SODA[13]. 一些工具比如 TXSPECTOR[14] 通过合约执行流程和交易记录联合分析，以及基于规则库辅助攻击检测；一些工具针对一些具体的漏洞. 例如，Nguyen 等人[15] 推出了 SGUARD，这是一种自动修复智能合约的工具，用以确保其不存在 4 个常见漏洞. SMARTPULSE 工具用于自动验证智能合约中的时间属性[16]. 一些工具使用程序分析方法对智能合约进行检测. 比如 EOSAFE 是第一个静态分析框架，旨在识别 EOSIO 智能合约中最常见的漏洞[17]. SMARTEST 是一种新颖的符号执行技术，用于有效识别智能合约中容易受攻击的交易序列[18]. EThor 是一种 EVM 字节码的自动静态分析器，是基于对 EVM 字节码语义的 Horn 子句抽象[19]. Slither 是一个用于自动检测漏洞和优化代码的静态分析框架[20].

另一方面，还已经有了一些关于 Depl 的研究. 大多数关于 Depl 的研究都集中在经济方面[21-25]. 只有少数论文强调了 Depl 面临的风险和挑战[22-26]，但没有深入探讨这些风险的原因和解决方案. Meier 等人[27] 建议使用技术接受和统一理论(UTAUT)来解决 Depl 贷款的合理性问题. Burda 等人[28] 研究了通过 Depl 中的代币向参与者分配决策权如何影响他们在平台启动之前和之后的角色.

与该工作最接近的工作是 TokenScope 和 TokenAware[29]. TokenScope 针对 ERC-20[30] 规则的代币进行了研究，重点解决了代币本身的代码功能与标准接口和标准事件不一致的问题. TokenAware 专注于定位复杂的核心数据结构来识别代币传输. 本文重点关注 Depl 期望的代币转账信息与代币实际转账信息的不一致，即行为不一致. 例如，如果 Depl 记录用户存入了 10 个代币，但用户实际转账的代币数量与此不同，则表明存在行为不一致.

尽管现有研究致力于智能合约的漏洞检测和修复，也有一些关于 Depl 的研究内容，但据我们所知，现有的智能合约研究都没有考察 Depl 和代币之间的不一致行为.

2 背景知识

本节将介绍一些与本文有关的相关概念.

1)账户(account). 以太坊由 2 种类型的账户组成——外部账户和合约账户. 外部账户由拥有相应私钥的个人控制，它无法作为程序执行或被调用. 合约账户由以太坊虚拟机(EVM)中执行的代码(智能合约)管理，它具备可执行的代码，也具备可以被调用的接口[31].

2)交易(transaction). 交易是指从一个帐户发送到另一个帐户的消息，这个消息中包含具体调用接口的名称和参数以及其他相关信息. 外部账户发起的交易称为外部交易，而执行过程中触发的其他交易称为内部交易. 每个外部交易都有一个区块链全局唯一的标识符，称为交易 ID 或交易哈希，其实质是一个长为 256 b 的索引.

3)全节点和全同步(full node & full sync). 以太坊中的全节点运行 EVM 实例，并通过同步维护区块链的相同副本. 当新节点加入以太坊网络时，它需要从其他节点下载区块并重放其 EVM 中的所有历史交易，以与现有节点达成共识. 这个过程称为完全同步[32].

4)追踪路径(trace). 在以太坊中，追踪路径是指从外部交易开始到完成的执行日志. 从 EVM 的角度来看，追踪路径是以太坊操作码的有序序列. EVM 按顺序执行这些操作码，生成合约数据变化的记录[33].

5)代币. 代币是一类智能合约，本质上是具有所有权账本、使用账本记录资产转移信息的智能合约的总称. 最被广泛采用的代币标准是 ERC-20 标准，它定义了 6 个代币的标准接口、2 个标准事件和所有权账本. 所有权账本的在代码中的表现形式即代币的核心数据结构. 标准接口用于代币传输，并将传输

信息存储在核心数据结构中. 2 个用于指导代币转移的标准接口方法分别是 Transfer 方法和 TransferFrom 方法[29].

6)去中心化交易所和去中心化借贷所(decentralized exchange & decentralized lending).去中心化交易所和去中心化借贷所是指通过智能合约实现的交易所和借贷平台. 我们将它们统称为 Depl. 用户可以与 Depl 合约进行交互, 由此在区块链上进行代币兑换或代币借贷, 实现代币在用户和平台之间的流动.

7)核心数据结构和行为(core data structure & behavior). 核心数据结构表示 Depl 或代币合约中记录代币转移信息的数据结构. 本文将代币和 Depl 的核心数据结构分别表示为 M 和 N, 将代币和 Depl 的行为分别表示为 B_m 和 B_n. 行为是指智能合约对合约自身的核心数据结构进行修改.

3 探索 Depl

本节将介绍 Depl 分析的结果, 4 位具有以太坊领域专业研究经验的研究人员参与了分析. 所有的信息都是从互联网上的各种渠道收集的, 通过自然语言处理的方式进行了分析, 并对分析结果进行了人工验证, 最终得到结果. 分析时间跨度从 2020 年开始跨越至 2022 年结束, 在 3 年的时间跨度范围内分析结果呈现一定不变性. 不变性表现在功能和核心数据结构基本不变. 这表明, 尽管区块链应用发展迅速, 但 Depl 在功能和代码实现上基本稳定, 代表该研究对未来 Depl 的发展具有一定的普适性意义.

据我们所知, 本文是第一篇研究和总结 Depl 如何存储代币转移信息的论文. 本节手工研究了所有白皮书和文档, 但发现它们几乎都没有考虑行为的不一致. 研究结果发现了一系列 Depl 共有的核心功能、大部分主流 Depl 都使用核心数据结构存储代币转移信息, 本文还定义了核心函数和核心事件, 并对核心数据结构的模式进行了分析.

3.1 分析范围

分析对象是从知名去中心化金融数据平台 Defillama 上获取的 2020—2022 年以太坊最流行的 60 个 Depl 应用. 所有信息均来自开源互联网. 信息种类包括 Depl 的白皮书、文档(或网页)和源代码. 源代码来自文档或 Github, 是进行分析时的最新代码.

3.2 核心功能分析

核心功能是指 Depl 为用户提供核心服务的功能. 同样也可以认为被用户使用次数最多的服务就是 Depl 的核心功能.

首先, 该工作利用自然语言处理的方式对白皮书、开源文档中的内容进行分析, 提取到一系列 Depl 的核心功能. 为了验证结果的准确性, 我们再针对自然语言处理的结果进行了人工的验证和补全. 同时, 该工作爬取 Etherscan 上这些合约最近 1 000 次被调用的接口中次数最靠前的函数. 分析发现最常见的功能有存款、取款、增加流动性、减少流动性、交易等.

值得注意的是, 自然语言处理领域在最近几年有显著的进步, 但不会对该工作的分析结果造成影响, 因为对处理结果都进行了人工核验和补全. 相反, 自然语言处理的进步会对核心功能分析的自动化提供更优越的辅助.

3.3 核心函数、核心事件、核心数据结构分析

虽然从文档和白皮书以及被调用次数中可以分析出应用层面 Depl 的核心功能, 但是从智能合约的角度出发, Depl 的内部实现仍未可知.

通过简单脚本的编写, 重点寻找既与核心功能在定义上较为相似的函数或者事件(即在命名或参数上相似), 又会在函数内部中调用代币传输接口的 Depl 外部函数(因为核心功能必然涉及到代币的转移). 这样做的目的是筛选那些可能实现核心功能的函数或标志核心功能被触发的事件.

4 位研究人员共同对上述脚本的结果进行了分析. 令人惊喜的是, 研究发现大多数最具代表性的 Depl 都使用核心数据结构来记录代币的转移信息. 实现核心功能并具备核心数据结构的函数称为核心函数, 标志核心功能被触发和被触发前也具备核心数据结构的时间称为核心事件.

表 1 是记录代币转移信息和不记录代币转移信息的 Depl 数量统计.

Table 1 Token Transfor Information Statistic Recorded by Depl

表 1 Depl 记录代币转账信息统计

记录数量	不记录数量	总数量
51	9	60

不记录代币转账信息的 Depl 主要用于 2 种场景: 1)Depl 只是传递订单, 起到起始点对点撮合的作用. 2)代币传输信息存储在链外, 仅中间传输信息存储在链上. 这些没有核心数据结构的 Depl 由于其功能的特殊性并不占据主流. 而主流 Depl 一般使用核心数据结构来存储代币转账信息.

主流的核心数据结构主要分为 3 类.

1)核心数据结构 1 是 *mapping*(*address*=>*struct*). 映射结构将代币的地址映射到代币的信息结构. 因此，这类核心数据结构 B_n 的语义是：值的增加表明 Depl 期望自己接收代币.

2)核心数据结构 2 是 *mapping*(*address*=>*mapping*(*address*=>*uint256*). 这个核心数据结构同时记录了正在交互的用户地址和代币地址. 这类核心数据结构 B_n 的语义是：值的增加表明 Depl 认为某一个账户向 Depl 转移了代币.

3)核心数据结构 3 是 *mapping*(*address*=>*mapping*(*address*=>*struct*)). 核心数据结构 3 与核心数据结构 2 具有相同的嵌套映射结构，但键值对中的值是结构体，结构体额外记录了除代币数量以外的其他信息. 结构体的语义与上面提到的 2 个核心数据结构是相同的.

除了以上 3 种核心数据结构外，还有一些特殊的核心数据结构. 例如，一些 Depl 各自使用自己定制的复杂数据结构. 这些核心数据结构的含义因 Depl 而异. 不同年度中的不同 Depl 各自拥有一些完全不同的核心数据结构，这些数据结构的出现和变化并不稳定，因此不认为它们是主流的核心数据结构.

3.4 模式分析

核心数据结构的描述在源码层面上进行，而模式是在字节码级别描述 EVM 如何定位核心数据结构的. 更具体地，模式是用于定位核心数据结构的非连续性操作码序列.

分析模式的目的是令后续工具和实验都具备从字节码层面定位 Depl 的核心数据结构的能力. 因为虽然在源代码中定位核心数据结构很简单，但大多数合约都不是开源的. 为了扩大智能合约实验的范围，本文将字节码作为研究对象，因为它在区块链上完全透明可获得，由此使实验范围可以扩大到已部署在以太坊上的合约整体.

例如，核心数据结构 1 对应的模式如图 1 所示，该模式使用用户的地址作为映射变量的键. SHA3 是对一类哈希算法提案的统称，在以太坊中的实现是 Keccak-256 算法，SHA3 可以将输入的内容散列为固定长度的输出[34]. SSTORE 是以太坊中的存储操作，该操作将存储值存储在存储地址中. SHA3 与键和核心数据结构 ID 的内容拼接并加上结构体偏移量得到的结果是 SSTORE 将存储新值的位置.

ID 是贮存变量(storage variable)的整数标识. 这个标识从 0 开始，按照源代码中定义的变量顺序依次递增. “地址|ID”代表拼接在一起后的结果. 箭头

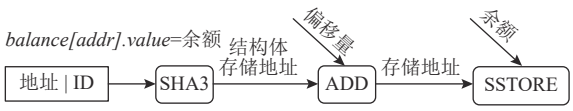


Fig. 1 Pattern 1 of Depl

图 1 Depl 的模式 1

的尾端代表被操作数，头部代表即将进行的操作. 当 SHA3 操作完成后，运算结果是一个 Storage 位置，这是区块链中信息持久化的单位. 此时的结果等于结构体变量的起始 Storage 位置. 对应在源码中是“*balance*[*addr*]”.

在以太坊的结构体实现中，结构体占用一个连续的存储空间，该空间被结构体项一一紧密地填满. 存放余额的变量可能处于结构体的任意一处. 因为 ADD 操作被用来更正偏移量，操作的结果即是余额最终的 Storage 位置.

核心数据结构 2 与核心数据结构 3 对应的模式分别如图 2 和图 3 所示. 模式 2 与模式 1 不同之处在于模式 2 将进行 2 次 SHA3 操作. 第 1 个键和核心数据结构 ID 的内容拼接作为第 1 次 SHA3 的操作数. 第 2 个键和第 1 次 SHA3 的结果拼接作为第 2 次 SHA3 的操作数. 第 2 次 SHA3 的结果是 SSTORE 存储新值的位置. 模式 3 则是模式 1 和模式 2 的结合，它既包含了 2 次 SHA3 操作，又具备了结构体的偏移操作.

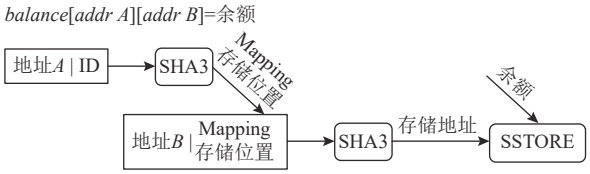


Fig. 2 Pattern 2 of Depl

图 2 Depl 的模式 2

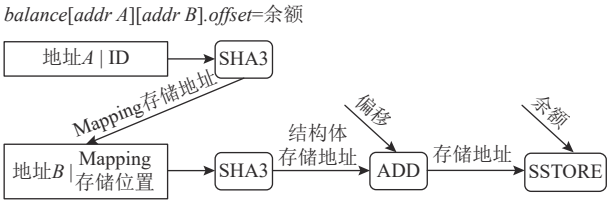


Fig. 3 Pattern 3 of Depl

图 3 Depl 的模式 3

4 DEALS

本节将介绍 DEALS，自动化检测 Depl 和代币行为之间不一致的工具的设计方式.

4.1 概 述

DEALS 将合约字节码和追踪路径作为工具的输

入，用以识别 Depl 行为和代币行为之间的不一致.

合约字节码和追踪路径作为以太坊区块链上公开透明的内容可以直接获取，获取的方法多样. 本文获得合约字节码的方式是使用谷歌提供的 BigQuery[35] 数据库，通过数据库查询的方法简便获取. 本文获取追踪路径的方式是通过全同步的方式从以太坊公链中获取.

一组不一致本质上是一个包含了不一致行为的交易，该交易中的不一致信息可以用 4 个字段来描述：TXID、Depl 地址、代币地址和不一致的代币数量.

如图 4 所示，DEALS 包括 3 个阶段：阶段 1，DEALS 定位不同合约的核心数据结构 ID. 阶段 2，以核心数据结构 ID 为基础，DEALS 使用全节点同步来重放所有交易. 阶段 3，DEALS 比较每条追踪路径中的 2 个行为，以找出不一致的实例.

阶段 1 的目的是定位合约中的核心数据结构. 只有在合约中定位到核心数据结构后，DEALS 才能监控其变化. 阶段 2 的目的是监视每个追踪路径中的 B_m 和 B_n. 由于每一条追踪路径都是以太坊上真实的操作留下的，检测每一条追踪路径意味着在真实的以太坊上检测不一致行为. 阶段 3 的目的是自动比较 B_m 和 B_n 的行为，最终找到每个追踪路径中包含的不一致之处. 综上，使用形式化的描述，即如果 DEALS 检测到 B_m 和 B_n 表示不同的代币转移信息，则 DEALS 认为该交易存在不一致.

4.2 阶段 1：定位核心数据结构

在智能合约中，每个贮存变量都有一个整数标识 ID. 这个整数表示从 0 开始，按照源代码中定义变量顺序依次递增. 由于用户余额信息作为永久信息存储在以太坊中，因此核心数据结构都是贮存变量. DEALS 使用这个唯一的 ID 来监视核心数据结构的行为. 因此阶段 1 的目标是寻找核心数据结构 ID.

寻找核心数据结构 ID 分为 2 个步骤进行：1）DEALS 根据 Depl 的模式获取数据结构 ID. 2）DEALS 对上一步骤获得的数据结构 ID 进行过滤，获得 Depl 的核心数据结构.

4.2.1 获取 Depl 数据结构 ID

DEALS 在步骤 1 中接收以太坊上所有已部署的字节码，并输出代币的核心数据结构 ID 和 Depl 的数据结构 ID.

TokenAware 是该工作的前驱工作，它可以被用来定位代币的核心数据结构. TokenAware 通过对代币智能合约账本的修改来推断代币的转移行为，并对由基本类型组成的复杂类型的指令序列进行定位. 借助 TokenAware，DEALS 可以轻松获取代币的核心数据结构 ID，但 TokenAware 不能直接在 Depl 中使用. TokenAware 的应用场景是代币，它的一些核心规则在 Depl 中并不成立.

具体来说，步骤 1 更改了 2 条 TokenAware 的核心规则. 第 1 条核心规则的更改是，在 TokenAware 中，需要固定匹配数据结构的修改次数为 2 次，而在 Depl 应用中规则为对数据结构的修改次数没有限制. 这是因为一笔代币可能在一笔 Depl 的交易中发生多次转账，因此 Depl 可以在一笔交易中多次记录代币转账信息. 第 2 条核心规则的更改是，模式的匹配需对应本文第 4 节中的 3 类 Depl 模式，而不是代币中的模式. 该规则用于避免过滤掉 3 类目标之外的无用数据结构.

总的来说，DEALS 利用了 TokenAware 的核心技术，并在这一步进行了改造. 最后得到代币的核心数据结构 ID 以及与 Depl 核心数据结构相同的数据结构 ID.

4.2.2 过滤 Depl 核心数据结构

在步骤 2 中，DEALS 需要将核心数据结构和其他具有相同字节码序列的不相关变量区分开来. DEALS

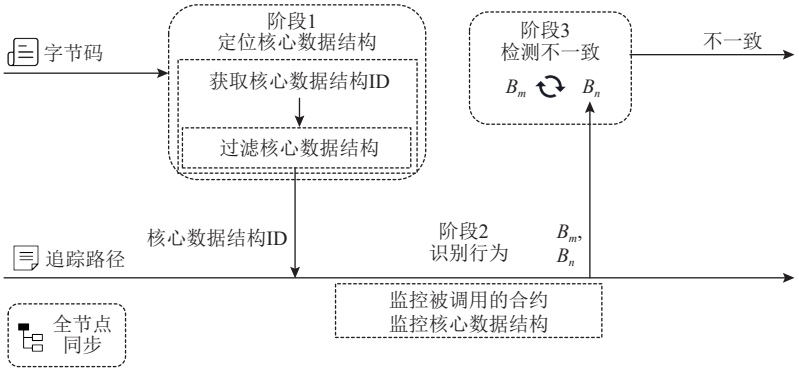


Fig. 4 Workflow of DEALS

图 4 DEALS 工作流程

使用 Depl 的核心函数和核心事件来过滤这些数据结构. 根据第 4 节的知识, 只有核心函数或发出核心事件的函数内部存储的数据结构才是存储代币转移信息的核心数据结构.

需要注意的是, 从 Depl 获取核心数据结构的源代码是从去中心金融平台 Defillama 在 2020—2022 年最流行的 60 个 Depl 的源代码. 该工作使用这些源代码来获取 Depl 的核心功能、函数、事件、数据结构, 是为了使这些信息更具有代表性. 由于以太坊的活跃性, 可能有很多非流行的 Depl 存在.

因此, 为了既使过滤条件具备代表性, 又要防止过度严格的过滤策略导致很多非流行的 Depl 的核心数据结构被过滤, 因此需要生成签名字典作为过滤条件.

签名字典以最流行的 60 个 Depl 的核心函数和核心事件字典为基准生成, 并混淆字典中函数和事件的参数顺序, 以减少参数对过滤算法的影响, 使得 Depl 的核心功能在过滤算法中成为主要的过滤条件, 又允许非流行的 Depl 的核心函数和事件的实现. 签名字典包括函数签名和事件签名. 例如, 函数签名是通过对函数名称和函数参数类型列表连接形成的字符串进行哈希处理并取前 4 个字节来生成的. 如果智能合约提供了这个函数接口, 那么函数签名就会出现在智能合约的原始字节码中.

在得到签名字典后, 具体的过滤算法如算法 1 所示.

算法 1. Depl 核心数据结构过滤算法.

输入: 字节码, 数据结构 ID;

输出: 核心数据结构 ID.

① while opcode do
② if PUSH4 then
③ *para* = *getPara*( );
④ if *para* in *FuncSigDic* then
⑤ if *DeplPattern* in *FuncOpcode* then
⑥ *Data Structure* → *Core Data Structure*;
⑦ end if
⑧ end if
⑨ end if
⑩ end while

算法 1 展示了通过函数签名字典过滤核心数据结构的过程. DEALS 按顺序过滤核心数据结构操作码. 当遍历到 PUSH4 操作码并且该操作码的参数在函数签名字典中(行②)时, 说明此时 DEALS 即将进入核心函数. 因为智能合约操作码采用了将函数签名压入栈的方式(即 PUSH4 函数签名)来标记函数的入口. 在核心函数内部, 符合 Depl 模式的数据结构就是核心数据结构.

DEALS 还记录合约地址和核心数据结构, 以供后续阶段使用.

4.3 阶段 2: 识别行为

在阶段 1 之后, DEALS 得到了代币和 Depl 的核心数据结构 ID. 阶段 2 利用核心数据结构 ID 在重放交易时识别它们的行为. 这个阶段分为 2 种情况: DEALS 首先监控交易中是否调用了 Depl 和代币合约, 如果同时调用了 Depl 和代币, 那么 DEALS 认为该笔交易存在 Depl 和代币的交互, 可能发生 Depl 和代币行为的不一致; 否则会继续监听交易过程中是否有任何行为, 并获取这些行为的相关信息.

4.3.1 监控被调用的合约

在一个追踪路径中可能涉及到调用多个合约, 但其中只有少数追踪路径会同时调用 Depl 和代币. 除了监控外部交易所调用的合约是否是 Depl 以外, DEALS 还在重放交易的过程中监视 CALL 操作码的执行, 从而监视被调用的合约列表.

当执行 CALL 操作码时, EVM 会提前在栈顶准备好 CALL 操作所需的参数. 必需的参数之一是调用的合约地址. DEALS 将在重放的过程中监控 Depl 和代币的合约地址是否都被调用. 只有在一条追踪路径中同时调用了 Depl 和代币时, DEALS 才会执行下一步操作. 这一步的目的是由于真实世界中大部分的交易并没有涉及到 Depl 和代币, 因此也不需要再去检测不一致的行为. 抛去了这些交易后将大大减轻 DEALS 的工作压力.

需要注意的是, 情况 2 是在满足情况 1 时, 分别在 Depl 和代币的执行语境中进行, 从而分别监控 Depl 和代币的核心数据结构和它们的行为.

4.3.2 监控核心数据结构

在情况 2 中, 通过模式 1~3, 可以找出具备这些不同核心数据结构模式的行为. DEALS 采用了维护污点地址表的方法, 其中加载了地址、存储位置和变化值的关系. 污点地址表的含义即被污点污染的地址的列表.

首先介绍构建污点表的过程. 初始状态具有 2 类污点源: 一是:将 Depl 合约的核心数据结构 ID 作为污点源; 二是将运行时所有的地址类型作为污点源.

污点传播有 2 类情况.

第 1 类情况是初始污点源向外传播的情况. 当执行 SHA3 操作码时, DEALS 会将 SHA3 操作码的 2 个操作数与正在执行合约的核心数据结构 ID 进行比

较，如果其中一个操作数匹配，并且另一个操作数是地址时，则将污点传播到运算结果且运算结果为Storage位置，并将该地址、核心数据结构ID和运算结果相关联. 污点的含义是它有可能与核心数据结构的行为有关，但并不能马上判断该数据与哪一个模式有关. 关联的意义是为了在找到行为后定位该行为与哪一个地址有关.

第2类情况是非初始污染源向外传播的情况. 如果SHA3操作或者ADD操作对污点表中一个非初始污点和一个新的地址进行操作，那么DEALS将SHA3操作结果标记为污点继续传播. 并且，将新的地址、此时的操作结果与非初始污点相关联的数据一同关联.

第1类污点传播可以将污点传播到模式1所在的Storage位置. 第2类污点传播可以将污点传播到模式2和模式3所在的Storage位置.

根据污点表中的数据，每当执行SSTORE操作码时，DEALS都会判断SSTORE的Storage位置是否在污点表中. 如果在污点表中，那么说明该SSTORE操作码改变的对象正是Depl的核心数据结构，意味着这个操作正在发生行为. 那么DEALS会以hook的方式先于SSTORE操作访问该Storage位置中的初始值，即余额原值，再访问SSTORE操作后该Storage位置的现值，即余额现值. 那么DEALS就得到了余额改变量. 同时通过污点表可以得到与该Storage位置相关的地址. DEALS检测存储位置的每次变化，并计算该追踪路径中所有变化的总量，由此得到了Depl或代币的行为.

4.4 阶段3：检测不一致

在阶段2，DEALS已经检测到了Depl和代币各自的行为，但是Depl与代币行为的不一致指Depl预期代币转移与代币本身的转移之间的差异. 因此需要对行为不一致在数据层面进行更加公式化的表达，从而使DEALS可以自动化检测不一致.

DEALS阶段3的结果如图5所示. 为了表述方便，将特定代币的地址和特定Depl的地址分别简写为*AoT*(address of Token)和*AoD*(address of Depl).

对于Depl的模式1而言，由于该模式直接使用映射结构作为Depl的核心数据结构，且映射结构的键是代币的地址，这意味着它只存储自己拥有的各种代币的余额. 那么B_n中三元组的含义是：*AoD 期待 AoT 将 value* 数量的代币转移给自己.

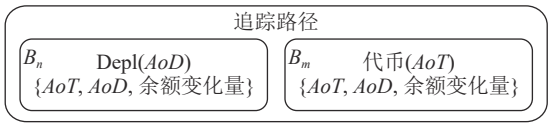


Fig. 5 Tuple result of DEALS in one trace

图5 一条追踪路径中DEALS的元组结果

对于Depl的模式2和Depl的模式3而言，由于模式2和模式3都具有嵌套映射的结构，因此它们可以准确地存储某个用户转移了某种类型的代币以及转移的金额. 那么B_n中三元组的含义是：某个用户将数量为*value*的*AoT*代币转移到了*AoD*中.

对于代币而言，B_m中的三元组的含义是该*AoT*将*value*数量的代币转移给了*AoD*. *AoT*实际上就是该代币自身的地址.

所以根据2类行为的语义，我们总结出了一条适用于自动化判断不一致行为的公式，即如果B_m和B_n中*value*不相同，则判定为发生了行为不一致. 行为不一致的代币的数量是两者*value*值相减的绝对值.

5 实　　验

5.1 实验结果

在第4节中，本文经过总共长达3年的理论调研分析，发现即使经过长时间的时间跨越，Depl的功能和代码实现仍然基本稳定. 为了检验该理论调研的正确性，我们选择了2018—2021年与2020—2022年时间跨度不同的年限作为实验的验证对象检验理论调研的正确性.

为了评估交易，实验重放了区块高度在5万~1 200万之间的交易，这些交易的生成时间为2018年1月30日至2021年3月8日，总共涉及8.59亿笔交易. 其中，有101万笔交易出现了不一致的行为，涉及110个Depl和2 544个代币，总共2 871组的Depl和代币. 实验结果被有序分为10类并公开在互联网[①]上.

所有结果在表2中展示.

DEALS总共检测到26 874个属于模式1的不一致、984 725个属于模式2的不一致以及1 150个属于模式3的不一致.

有902 191笔不一致的交易发生在与地址为0x2a0c[②]的Depl交互过程中，这是结果中发生过最多

① https://github.com/R-K-Jiang/DEALS

② 0x2a0c0DBEcC7E4D658f48E01e3fA353F44050c208

Table 2 Relevant Statistical Data of Inconsistent Depl

表 2 发生不一致的 Depl 的相关统计数据

Depl 模式	模式 1	模式 2	模式 3
Depl	3 024/3 026	434/911	7/21
不一致的 Depl 种类	53/53	45/55	2/2
相关联的代币种类	59/59	1 594/1 783	982/983
不一致的交易数量	26 874/26 874	978 345/984 725	1 150/1 150

注: "/"前的数字表示开源 Depl 以及与其相关数据的数量, "/"后的数字表示所有 Depl 以及与其相关数据的数量.

不一致交易的 Depl. 在地址为 0x2a0c 的 Depl 与地址为 0x4a22[①]的代币交互过程中, 总共有 35 499 笔交易存在不一致, 这是所有检测出不一致最多的 Depls 和代币组. 由于地址长度太长, 本文只给出地址的前 4 位数字.

5.2 DEALS 的精准度

为了解析 DEALS 的精准度, 我们手动检查了所有检测出不一致的 Depl 和代币组, 总共 2 871 组. 误报的含义是, 当 Depl 与代币实际没有发生行为不一致, 但 DEALS 检测到不一致时即为误报. 经结果分析, 仅发现 57 个误报, 这意味着 DEALS 的精准度达到 98.0%.

误报主要有 2 个原因: 1)部分用户不直接与 Depl 交互, 而是调用私有部署的中间合约间接与 Depl 交互. 这些中间合约的用处是简化用户需要多次才能完成的操作. 但 Depl 只记录中间合约的余额, 但中间合约又将代币转移给实际所有者. 8 对 Depl 和代币以及 1 237 个不一致的误报都属于这种情况. 2)部分 Depl 在充值或提现时可以指定其他地址而不是 *msg.sender* 作为充值账户, 这导致 DEALS 检测到代币的核心数据中 *msg.sender* 对应的余额减少, 但 Depl 的核心数据结构中 *msg.sender* 对应的余额并没有增加. 49 对 Depl 和代币以及 612 个不一致的误报都属于这种情况.

Depl 和代币这 2 类误报都是由于代币实际转入的账户与 Depl 的存款账户不同造成的. 因此, 如果一条追踪路径中存在另一个账户的代币余额变化等于 Depl 的存款账户的余额变化, 那么该结果可能属于误报. 但由于其他不一致也可能具备与误报相似的特征, 为了减轻误报可能导致损失更多真实的不一致, 并且该类情况相对较少, 因此该类误报处于可以承受的范围.

另外, 如果 Depl 在其官网或白皮书中已经声称存在不一致, 那么这些不一致可以被视为不会造成恶劣影响的不一致, 因为用户会提前知晓这种风险. 对此, 我们手动检查了每个 Depl 是否都有其官方网站或白皮书, 但不幸的是, 很少有 Depl 描述不一致, 这可能会导致安全隐患或造成经济损失.

6 行为不一致的原因

将误报过滤后, 总共有 2 819 组 Depl 和代币. 通过手动分析, 本文确定了不一致的 10 个主要原因. 如表 3 所示, 这 10 类主要原因的分类标准是根据获利目标、受害对象、是否是金融策略、是否具有误导性、故意实现或无意实现, 同时结合代码的语义进行了人工判断. "获利目标"指当不一致发生时可能获利的对象; "受害对象"指当不一致发生时可能受害的对象; "是否是金融策略"指该不一致的成因是否是金融策略导致的; "是否具有误导性"指是否有误导性的操作导致不一致或该不一致是否导致用户的困惑; "故意实现或无意实现"指该不一致的成因是 Depl 代码有意编写所形成的还是无意形成的漏洞.

不一致的原因概括展示在表 4 中.

6.1 诈骗 Depl

出现这种不一致原因的 Depl 数量为 4 种, 共有 68 笔交易产生不一致, 导致 17 种因该原因造成的代

Table 3 Classification Criteria of Inconsist Causes

表 3 不一致成因的分类标准

分类标准	诈骗 Depl	未经检查的代币	重入漏洞	整数溢出	尺度不一致	Depl 的收费政策	锁币规则不明确	Depl 的奖励政策	Depl 的待办机制	第三方投资
获利目标	Depl	Depl	黑客	黑客	Depl 或黑客	Depl	Depl	用户	无	Depl
受害对象	用户	用户	Depl	Depl	Depl 或用户	用户	用户	无	无	用户
是否是金融策略	否	否	否	否	是	是	是	是	是	是
是否具有误导性	是	否	否	否	是	是	是	是	是	是
故意实现或无意实现	故意	无意	无意	无意	无意	故意	无意	故意	故意	故意

① 0x4a220E6096B25EADb88358cb44068A3248254675

Table 4 Reasons of Inconsistency

表 4 不一致的原因

原因类型	Depl 种类数量	原因描述
诈骗 Depl	4	Depl 偷走了所有存款
未经检查的代币	38	Depl 不检查代币是否按照标准接口实现功能
重入漏洞	1	Depl 存在重入漏洞
整数溢出	1	Depl 存在整数溢出漏洞
尺度不一致	25	Depl 和代币在账本中的尺度不一致
Depl 的收费政策	32	当用户使用 Depl 进行访问操作时，Depl 向用户收取费用
锁币规则不明确	1	由于未公开的代币扣押政策导致用户代币丢失
Depl 的奖励政策	4	Depl 因利息而向用户转移更多代币或记录更多代币
Depl 的待办机制	3	Depl 使用两个关联的数据结构存储用户余额.
第三方投资	1	Depl 通过生成凭证代币等间接存储用户余额.

币损失. 在这类原因中, 用户存入 Depl 的代币的所有权不是自己的, 而是管理员指定的地址. 这意味着用户如果不是所有者, 则只能将代币存入 Depl, 而不能提取代币. 我们怀疑 Depl 是诈骗合约, 因为用户无法取回存入此 Depl 的代币.

图 6 显示了诈骗 Depl. 合约所有者可以调用 *changeFounder* 来指定代币记录的地址. 当用户调用 *depositToken* 并将其令牌转移到 Depl 时, Depl 实际上记录了该令牌属于 *FounderAddress*. 而如果用户想要提取代币, Depl 断言, 由于记录中的代币属于 *FounderAddress*, 因此用户无权将其取走.

```
1 function depositToken(address _token, uint256 _amount){
2 ......
3   _token.transferFrom(msg.sender, this.address, _amount);
4 ......
5   tokens[_token][founderAddress] +=_amount;
6 }
7 function changeFounder(address _newAddress){
8   require msg.sender == owner;
9   founderAddress = _newAddress;
10 }
```

Fig. 6 Cheat Depl

图 6 诈骗 Depl

6.2 未经检查的代币

出现该不一致原因的 Depl数量为 38 种, 共有 29 788 笔交易产生不一致, 导致 415 种因该原因造成的代币损失. 出现该类不一致的原因是 Depl 默认代币的行为与标准函数表达的含义相同, 并且没有对代币进行安全检查, 从而导致不一致. 如图 7 所示, Depl 认为它接收到的代币数量就是标准函数 *tokensToTransfer* 中参数的数量, 但在该类原因中, 代币的行为与标准函数表达的含义并不一致. 这使得 Depl 被代币欺骗, 造成用户和 Depl 的损失. 代币的行为与标准函数表达的含义不一致的方式有很多, 包括但不限于代币收取手续费用、代币不合规范的铸造、代币销毁等. 例如, 代币具有燃烧机制. Depl 认为令牌将 *value* 传输到 *to*. 但实际上 *to* 收到 *tokensToTransfer*, 它等于 *to* 减去 *tokensToBurnAndMint*. 已经有针对此类不一致的研究, 因此不再在本文中详细解释它们.

```
1 function transfer(address to, uint256 value) public returns(bool){
2   uint256 tokensToBurnAndMint = findOnePercent(value);
3   uint256 tokensToTransfer = value.sub(tokensToBurnAndMint);
4 ......
5   _balances[msg. sender] = _balances[msg.sender].sub(value);
6   _balantes[to] = _balances[to].add(tokensToTransfer);
7   _totalSupply = _totalSupply.sub(tokensToBurnAndMint);
8 ......
9 }
```

Fig. 7 Unchecked burning mechanism

图 7 未经检查的燃烧机制

6.3 重入漏洞

出现这种不一致原因的 Depl 数量为 1 种, 共有 548 笔交易不一致, 导致 2 种因该原因造成的代币损失. DEALS 可以检测由于重入漏洞而导致的不一致. 从不一致的角度来看, 漏洞的发生导致黑客的代币核心数据结构余额激增, 但 Depl 的核心数据结构余额仅小幅度下降. 该漏洞产生的原因是: 1)Depl 没有添加重入限制; 2)ERC777 标准的代币无法控制回调的合法性.

6.4 整数溢出

出现该不一致原因的 Depl 数量为 1, 总共有 1 笔交易不一致, 导致 1 种因该原因造成的代币损失. Depl 通常使用 uint256 类型来存储用户的余额, 这是最后存储在 Depl 模式中并输入到存储中的类型. uint256 是一个长度为 256 b 的无符号数, 也是 EVM 中最大的整数类型. 当计算结果大于 2^{256}, 也就是 uint256 可以存储的最大值, 可能造成正溢出, 从而导致核心数据结构存储一个比正常结果更小的值. 如果计算结果是小于 0 的数, 即计算结果下溢, 就会导致核心数据结构存储一个非常大的正整数. 如果一个 Depl 存在整数溢出漏洞, 则其合约中可能不会包含或使用 *safemath* 库, 从而导致该合约中出现整数溢出现象. 该溢出漏洞可能导致核心数据结构记录错误, 对 Depl 造成严重的财产损失.

6.5 尺度不一致

出现这种不一致原因的 Depl 数量为 25 个. 共有 6 814 笔交易不一致, 导致 26 种因该原因造成的代币

损失. Depl 和代币各自账本的规模不一致会导致记录不一致. 由于没有标准规定合同需要使用的规模, 因此合同核算所使用的规模是不确定的. 如图 8 所示, *atomValue* = *value.mul*(*atomsPerMole*)才是实际的值而不是参数 *value*. 这种尺度的不一致最终导致行为的不一致.

```
1 function transfer(address to, uint256 value) public returns(bool){
2 ......
3   uint256 atomValue = value.mul(atomsPerMolecule);
4   atomBalances[msg.sender] = atomBalances[msg.sender].
    sub(atomValue);
5   atomBalances[to] = atomBalances[to].add(atomValue);
6 ......
7 }
```

Fig. 8 Scale inconsistency

图 8 尺度不一致

如果代币设计者考虑到该种情况, 则有可能避免财产损失, 但是仍然会让用户感到相当困惑. 因为 Depl 会自定义小数尺度, 但对于被最广泛使用的钱包 MetaMask[36] 来说, 其代币的小数尺度默认为 0, 这意味着用户在与此类 Depl 交互时, 可能会发现钱包显示转入 Depl 的金额与 Depl 界面向用户显示的余额不同, 造成用户对于资金去向的恐慌.

更需要注意的是, 如果该种不一致情况没有被正确处理, 部分合约实际使用中有可能导致财产损失. 例如, 图 9 显示了与总供应量相关的规模, 即 *atomsPerMolecule* 是可以被人为控制的. 假设 *atomsPerMolecule* 一开始是 10, 如果用户在 Depl 中存入 100 个代币, 那么 Depl 记录了该用户拥有的 100 个代币. 但是根据图 8 中的第 4 行和第 6 行, Depl 收到的代币记录为 1 000 个. 然后 *atomsPerMolecule* 更改为 100. 用户仍然可以合理地要求 Depl 转帐 100 给自己, 但实际上, 这个操作需要 Depl 在代币中的记录达到 10 000 个. 一个危险的假设是, 黑客在 *atomsPerMolecule* 较低时向 Depl 存入大量代币, 然后恶意控制 *atomsPerMolecule* 变高. 这时黑客要求 Depl 提取押金, Depl 不会认为这样的操作是错误的, 从而向黑客大量转移代币, 造成 Depl 的财产损失. 同时, 因为这大大减少了代币中 Depl 的余额记录, 当其他用户想要提现时, 交易会因为 Depl 没有足够的余额而被还原. 这相当于窃取了其他用户在 Depl 的存款, 导致 Depl 和用户的财产都受到重大威胁.

```
1 function rebase(uint256 supplyDelta, bool increaseSupply) external
  returns (uint256){
2 ......
3   atomsPerMolecule = totalAtoms.div(totalSupply);
4 ......
5 }
```

Fig. 9 Rebase

图 9 重复基底

6.6 Depl 的收费政策

出现这种不一致原因的 Depl 数量为 32 个. 总共有 947 002 笔交易出现不一致, 导致 2 265 种因该原因造成的代币损失. Depl 的收费政策是指操作的 Depl 对于用户进行金融活动的行为收取手续费用. 这些操作在智能合约层面来讲是一些功能函数的调用, 例如充值代币、提现代币、兑换代币等. 对于充值代币而言, Depl 可以通过减少 Depl 账本中用户存款的记录来收取费用, 这意味着实际用户支付的代币比账本中的多. 对于提现代币而言, Depl 可以通过转移比用户请求的金额更少的代币来收取费用, 这意味着如果 Depl 账本显示用户有 *value* 金额的代币存款, 则用户实际可以提款的金额少于 *value* 个代币. 对于兑换代币而言, 如果用户想要将代币 *A* 换成代币 *B*, Depl 可以选择是取代币 *A* 或代币 *B* 作为费用.

只有 2 个 Depl 在它的官方网站、文档或者白皮书中详细解释了收费和收费比例, 5 个 Depl 使用某些金融公式动态地计算收费的金额, 其余 25 个 Depl 则没有相关的说明.

如图 10 所示, 第 7 行实际记录的金额比用户在存款操作中的少. 收取的比例可能因合约而异, 但无论以何种方式收取多少比例的手续费, 如果用户不知道扣款操作, 最终都会导致用户的财产损失和困惑.

```
1 function depositToken(address token, uint amount) public {
2   require (token ≠ address(0));
3   require (isTokenActive(token));
4   require(Token(token).transferFrom(msg.sender, address(this),
    amount));
5   uint feeDepositXfer = amount.mul(feeDeposit[token])/(1 ether);
6   uint depositAmount = amount.sub(feeDepositxfer);
7   tokens[token][msg.sender] = tokens[token][msg.sender]
    .add(depositAmount);
8   tokens[token][feeAccount] = tokens[token][feeAccount]
    .add(feeDepositXfer);
9   emit Deposit(token, msg.sender, amount, tokens[token]
    [msg.sender]);
10 }
```

Fig. 10 Fee policy of Depl

图 10 Depl 的收费政策

6.7 锁币规则不明确

出现该不一致原因的 Depl 数量为 1 个. 共有 3 笔交易不一致, 导致 2 种因该原因造成的代币损失. 这种类型的 Depl 会存储一个代币白名单. 只有当指定的代币在白名单中时, 用户才可以交易该代币, 否则, 所有存入 Depl 的该类型代币将被 Depl 收集, 并

且无法再提现. 如图 11 的第 15 行, 该函数指定了某一类代币是否为 *active*. 根据第 3 行, 只有 *active* 的代币可以被交易. 并且根据第 13 行, *Agent* 可以随时更改其状态, 而无需对任何其他人员进行通知或者有任何其他限制, 这将具有极大的风险.

```
1 function depositToken(address token, uint amount) external {
2   require(token ≠ address(0));
3   if (whitelistTokens[token].active) {
4   require (whitelistTokens[token].timestamp <= now);
5   require (ERC20I(token).transferFrom(msg.sender, this, amount));
6   tokens[token][msg.sender] = safeAdd(tokens[token]
    [msg.sender], amount);
7   emit Deposit(token, msg.sender, amount, tokens[token]
    [msg.sender]);
8 }else{
9   require(ERC20I(token).transferFrom(msg.sender, this, amount));
10 tokens[token][feeAccount] = safeAdd(tokens[token][feeAccount],
    amount);
11 emit PayFeeListing(token, msg.sender, amount, tokens[msg.sender]
    [feeAccount]);
12 }
13 function setWhitelistTokens(bool active, .....) external onlyAgent {
14 .....
15 whitelistTokens[token].active = active;
16 .....
17 }
```

Fig. 11 Activate mechanism at will

图 11 任意的激活机制

由于 *active* 符号是由合约代理设置的, 对普通用户并不公开, 并且这个 Depl 没有相关文档对其进行解释. 除非用户是专业的以太坊研究人员, 并且可以直接访问公链上的 Depl, 否则用户很大概率会丢失代币.

6.8 Depl 的奖励政策

出现这种不一致原因的 Depl 数量为 4 个. 共有 22 403 笔交易不一致, 导致 15 种因该原因造成的代币损失. 存在此类不一致的合约一般属于投资或借贷合约. 此类合约的主要功能是向用户提供贷款并收取还款利息. 贷款的代币并不属于 Depl 本身所有, 而是由投资者投资. 当投资者希望提取资产时, 他们可以获得股息. 一般来说, 股息实际上是从贷款利息中获得的. 这些奖励的金额是通过复杂的财务计算得来的, 这与存款的时间和 Depl 的运作情况有关. 奖励结算的方式可以是 Depl 在账本中记录更多的余额, 也可以自动向用户转移更多的代币. 一般来说, 虽然这种类型的合约会造成不一致, 但对用户来说是无害的.

6.9 Depl 的待办机制

存在这种不一致的 Depl 数量为 3 个. DEALS 检测到不一致的原因是此类 Depl 使用 2 种数据结构而非 1 种数据结构来存储用户余额. 这 2 种数据结构是核心数据结构和“待办”结构. 如图 12 所示, 在存款操作中, 本次存款的金额存储在“待办”结构中, 而之前存款的金额则从“待办”结构转移到核心数据结构(第 1 行). 所以用户的总存款金额应该是核心数据结构和“待办”结构的总和. 从用户的角度来看, 由于 Depl 实现了 2 种结构体的封装, 因此在查看余额和提款时, 他们不会注意到这与正常操作有什么区别. 与大多数 Depl 的核心数据结构模式不同, 因为这种操作以一种非常新颖的方式处理用户余额. 我们将在以后的工作中将深入探讨这种多种数据结构共同决定用户余额的 Depl.

```
1 function deposit(address token, uint256 amount) public {
2   updateDepositsBalance(msg.sender, token);
3   SafeERC20.safeTransferFrom(IERC20(token), msg.sender,
    address(this), amount);
4   balanceStates[msg.sender][token].pendingDeposits.amount
    = balanceStates[msg.sender][token].pendingDeposits.amount
    .add(amount);
5   balanceStates[msg.sender][token].pendingDeposits.batchId =
    getCurrentBatchId();
6   emit Deposit(msgsender, token, amount, getCurrentBatchId());
7 }
```

Fig. 12 Pending mechanism of Depl

图 12 Depl 的待办机制

6.10 第三方投资

出现该不一致原因的 Depl 数量为 1. 共有 4 笔交易不一致, 导致 2 种因该原因造成的代币损失. Depl 的记录不是用户存入的代币数量, 而是“ctoken”的数量. Ctoken 是 Compound 中的一种资产凭证, 在用户向 Compound 中存入财产时自动生成. Compound 是一种借贷合同, 向其中存入财产可以获得利息. 此类 Depl 实际上本身没有金融架构, 而是单纯将用户存入的代币转投进入 Compound 合约中.

Ctoken 和代币的交换并不是一对一的, 而是视 Compound 合约的具体协议而定, 这导致 Depl 记录的数量和代币记录的数量不一致. 在图 13 的第 7 行中, Depl 接收用户的存款代币, 并紧接着在第 8 行将代币借给 Compound 并接收 Ctoken. 由于汇率的原因, *ctokenAmt* 不一定等于 *_amt*. 所以在第 11 行, Depl 与代币的记录不同.

这种情况非常有趣, 因为该种 Depl 其实是一个 Depl 空壳, 它的存在严重依赖于底层 Depl. 如果该类 Depl 有清晰的文档说明, 则不存在安全问题, 但遗憾的是已经无法再在互联网上寻找到该 Depl 任何可用或可访问的信息, 因此该类 Depl 的实际情况也无从可知.

7 案例讲解

1)地址为 0xc71A[①]的存在下溢出漏洞的 Depl. 在

```
1 function deposit(address token, uint amt) external payable returns
  (uint _amt) {
2 ......
3   address cErc20 = tknToCTkn[token];
4   uint initalBal = tokenBal(cErc20);
5 ......
6   _amt = amt == (uint(−1))? IERC20(token).balance0f
    (msg.sender):ant;
7   IERC20(token).safeTransferFrom(msg.sender, address(this), _amt);
8   require(CTokenInterface(cErc20).mint(_amt) == 0, "mint-failed");
9 ......
10   uint finalBal = tokenBal(cErc20);
11   uint ctokenAmt = sub(finalBa1, initalBal);
12   liquidityBalance[token][msg.sender] += ctokenAmt;
13 ......
14 }
```

Fig. 13 Investment assets

图 13 投资资产

区块高度为 8 117 422 处 TXID 为 0xa551 的交易受到了攻击. 如图 14 所示, 第 4 行存在下溢漏洞. 当 *tokens*[*token*][*msg.sender*]<*amount* 时, 就会发生下溢, 核心数据结构实际上在 Depl 的记录中凭空存储了大量的代币. DEALS 通过比较其核心数据结构变化的不一致情况来检测未报告的攻击事件. DEALS 检测结果显示, B_n={0xc71A,0xd79e,1.157e+77}, 而 B_m={0xc71A, 0xd79e,5e+11}. 编号为 0xa551 的交易由地址为 0xd79e 的用户发送. 交易前, Depl 账本显示地址为 0xd79e 的用户的存款为 0, 但 Depl 没有检测到余额大小, 导致计算下溢.

```
1 function withdrawCollateral(address token, uint amount) public {
2 ......
3   uint amountToWithdraw = amount;
4   tokens[token][msg.sender] = tokens[token][msg.sendesr] - amount;
5   emit Withdraw(token, msg.sender, amountToWithdraw,
    amountToWithdraw);
6   require(
    StandardToken(token).transfer(msg.sender, amountToWithdraw),
    "error with transfer");
7 ......
8 }
```

Fig. 14 Overflow vulnerability

图 14 溢出漏洞

2)地址为 0xBDFB[①]的诈骗 Depl. 案例以一个不一致的交易来说明该诈骗 Depl 是如何进行诈骗的. 对于不一致交易 0xce1f 而言, DEALS 检测到 B_n 为 {0xf2EA, 0xBDFB,0}, 而 B_m 为 {0xf2EA, 0xBDFB, 10e+19}. B_m 表示用户向 0xBDFB 存入了数量为 10e+19 的代币 0xf2EA, 而 B_n 表示 Depl(0xBDFB)并没有认可该代币的存入.

该交易的实际流程是用户 0xc419 确实将 10e+19 个代币 0xf2EA 转移到 0xBDFB. 但由于图 6 的第 5 行, 该 Depl 恶意记错账本, 将本应认可的代币存入记录记为他人存入, 否定了用户 0xc419 的转账行为. 这种行为实际是利用管理员权限诱骗用户存钱. 最后该 0xBDFB 带着赃款离开.

3)地址为 0xEc3D[②]的锁币规则为不明确的 Depl. 该 Depl 总共导致超过 1 亿枚代币被罚而无法取出. 案例以一个不一致的交易来说明该 Depl 不明确的锁币规则诈骗. 对于交易 0x6d8a 而言, DEALS 检测到 B_n={0xC132, 0xEc3D,0}, 而 B_m={0xC132, 0xEc3D, 1.3e+8}. B_m 表示用户向 0xEc3D 存入了 数量为 1.3e+8 的代币 0xC132, 而 B_n 表示 0xEc3D 并没有认可该代币的存入.

该交易的详细流程是用户 0x1289 向 0xEc3D 存入了数量为 1.3e+8 的代币 0xC132, 但由于图 11 的第 10 行和第 15 行, 0xEc3D 将没有被“激活”的代币存入了 *feeAccount*, 从而否认了用户 0x1289 的存入行为, 认为没有收到记录存入了 10^7 的代币 0xC132. 最终导致用户 0x1289 将永远无法提取存入 0xEc3D 中的代币.

8 结论与展望

Depl 与代币的不一致会导致 Depl 或用户财产的损失或者用户对 Depl 的不信任. 本文提出 DEALS, 利用代币的核心数据结构行为和 Depl 的核心数据结构行为来检测它们之间是否存在不一致. 通过检测五百万到一千二百万个区块上的所有交易, DEALS 检测到 1 012 749 笔不一致交易, 涉及 2 876 对 Depl 和代币, 涉及 110 个 Depl 和 2 571 个代币. 通过手动分析, 本文总结了 10 类不一致的原因.

由于功能的复杂性, DEPL 有时不能仅使用单个数据结构来表示分类帐记录. 如果 Depl 使用复杂的财务公式来实施复杂的交易, 可能导致 DEPL 不再使用单个数据结构来存储用户分类帐, 而是使用多合同的关节存储或多 DATA 结构的关节存储. 并且 DEALS 在获得结果后需要人工分析并对不一致原因作出总结, 这些工作的任务量是庞大和耗时的. 这些目标均具备一定的挑战性, 因此值得一篇新的文章来进行讨论和解决. 未来我们将在这些目标中继续跟进.

作者贡献声明: 姜人楷提出了算法思路和实验方案并撰写论文; 宋书玮负责完成实验和撰写部分

① 0xBDFBa805cC6482070485dfDe1F9D6508F7709C35

② 0xEc3D7968b0D3FFF0A074668E08EB56c5e6d38B21

论文；罗夏朴和陈厅给出指导意见并修改论文；罗瑞杰、王炳森和乔翱负责完成部分实验以及部分实验结果的人工分析.

参 考 文 献

[1] Junsang K, Seyong K. A survey of decentralized finance (DeFi) based on blockchain[J]. Journal of the Korea Society of Computer and Information, 2021, 26(3): 59−67

[2] DefiLlama.Totalvaluelockedrankings[DB/OL].[2022-09-12].//https://defillama.com/

[3] Ethereum.Smartcontract[EB/OL].[2022-09-12].https://ethereum.org/en/developers/docs/smar t-contracts/

[4] Ethereum. Ethereum eips[EB/OL]. [2022-09-12].https://github.com/ethereum/EIPs

[5] Etherscan. Token tracker[EB/OL]. [2022-09-13].https://etherscan.io/tokens

[6] DefiLlama. Total value locked rankings of Ethereum[DB/OL]. [2022-09-14].https://defillama .com/chain/Ethereum

[7] Chen Ting, Zhang Yufei, Li Zihao, et al. TokenScope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum[C]// Proc of the 2019 ACM SIGSAC Conf on Computer and Communications Security. New York: Association for Computing Machinery, 2019: 1503−1520

[8] Enzyme Finance. Oyente[EB/OL]. [2022-09-15].https://github.com/enzymefinance/oyente

[9] Trail of Bits. Manticore[EB/OL]. [2022-09-16].https://github.com/trailofbits

[10] So S, Lee M, Park J, et al. VeriSmart: A highly precise safety verifier for Ethereum smart contracts[C]//Proc of 2020 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2020: 1678-1694

[11] MythX. MythX[EB/OL]. [2022-09-17].https://mythx.io/

[12] Rodler M, Li Wenting, Karame G O, et al. Sereum: Protecting existing smart contracts against re-entrancy attacks[C] // Proc of ISOC Network and Distributed System Security Symp. San Diego, CA: The Internet Society, 2019

[13] Chen Ting, Cao Rong, Li Ting, et al. SODA: A generic online detection framework for smart contracts[C]// Proc of ISOC Network and Distributed System Security Symp. San Diego, CA: the Internet Society, 2020: 23-26

[14] Zhang Mengya, Zhang Xiaokuan, Zhang Yinqian, et al. TXSPECTOR: Uncovering attacks in Ethereum from transactions[C]//Proc of 29th USENIX Security Symp (USENIX Security'20). Berkeley, CA: USENIX Association, 2020: 2775-2792

[15] Nguyen T D, Pham L H, Sun Jun. SGUARD: Towards fixing vulnerable smart contracts automatically[C]//Proc of 2021 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2021: 1215-1229

[16] Stephens J, Ferles K, Mariano B, et al. SMARTPULSE: Automated checking of temporal properties in smart contracts[C]//Proc of 2021 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2021: 555-571

[17] He Ningyu, Zhang Ruiyi, Wang Haoyu, et al. {EOSAFE}: Security analysis of EOSIO smart contracts[C]//Proc of 30th USENIX Security Symp (USENIX Security'21). Berkeley, CA: USENIX Association, 2021: 1271-1288

[18] So S, Hong S, Oh H. SMARTEST: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution[C]//Proc of 30th USENIX Security Symp (USENIX Security'21). Berkeley, CA: USENIX Association, 2021: 1361-1378

[19] Wüst K, Matetic S, Egli S, et al. ACE: Asynchronous and concurrent execution of complex smart contracts[C]// Proc of the 2020 ACM SIGSAC Conf on Computer and Communications Security. New York: Association for Computing Machinery, 2020: 587-600

[20] Feist J, Grieco G, Groce A. Slither: A static analysis framework for smart contracts[C]//Proc of 2019 IEEE/ACM 2nd Int Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). Piscatway, NJ: IEEE, 2019: 8-15

[21] Zetzsche D A, Arner D W, Buckley R P. Decentralized finance[J]. Journal of Financial Regulation, 2020, 6: 172−203

[22] Chen Yan, Bellavitis C. Blockchain disruption and decentralized finance: The rise of decentralized business models[J]. Journal of Business Venturing Insights, 2020, 13: e00151

[23] Werner S, Perez D, Gudgeon L, et al. Sok: Decentralized finance (defi)[C]// Proc of the 4th ACM Conf on Advances in Financial Technologies. New York: Association for Computing Machinery, 2022: 30-46

[24] Qin Kaihua, Zhou Liyi, Afonin Y, et al. CeFi vs. DeFi--Comparing centralized to decentralized finance[J]. arXiv preprint, arXiv: 2106.08157, 2021

[25] Jensen J R, von Wachter V, Ross O. An introduction to decentralized finance[J]. Complex Systems Informatics and Modeling Quarterly, 2021(26): 46−54

[26] Popescu A D. Decentralized finance–the lego of finance[J]. Social Sciences and Education Research Review, 2020, 7(1): 321−349

[27] Meier M, Mattke J, Maier C. Decentralized finance: A configurational perspective on UTAUT[J]. European Conference on Information Systems, 2022(102): 1577−1589

[28] Burda M, Locca M, Staykova K. Decision rights decentralization in DeFi platforms[J]. European Conference on Information Systems, 2022(145): 1826−1839

[29] He Zheyuan, Song Shuwei, Bai Yang, et al. TokenAware: Accurate and efficient bookkeeping recognition for token smart contracts[J]. ACM Transactions on Software Engineering and Methodology, 2023, 32(1): 1−35

[30] IERC-20. OpenZeppelin[EB/OL]. [2022-09-22].https://github.com/OpenZeppelin/openzeppelincontracts/blob/master/contracts/token/ERC20/IERC20.sol

[31] Ethereum. Account types[EB/OL]. [2022-09-20].https://ethereum.org/en/developers/docs/acco unts/

[32] Ethereum. Full node[EB/OL]. [2022-09-19].https://ethereum.org/en/developers/docs/nodes-and-clients/#full-node

[33] GAVIN WOOD. Ethereum: A secure decentralised generalised transaction ledger[DB/OL]. [2022-09-12].https://ethereum.github.io/yellow-

paper/paper.pdf
[34] Ethereum. SHA-3[EB/OL]. [2022-09-15].https://ethereum.org/zh/developers/docs/consensus-mechanisms/pow/mining/mining-algorithms/ethash/#sha3
[35] Google. BigQuery[EB/OL]. [2022-09-15].https://cloud.google.com/bigquery
[36] MetaMask. The crypto wallet for defi, web3 dapps and nfts — metamask[EB/OL]. [2022-09-19].https://metamask.io/

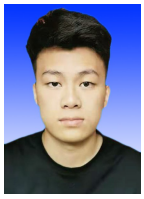
Jiang Renkai, born in 1999. Master candidate. His main research interest includes blockchain security.
姜人楷, 1999 年生. 硕士研究生. 主要研究领域为区块链安全.

Song Shuwei, born in 1999. PhD candidate. His main research interest includes blockchain security.
宋书玮, 1999 年生. 博士研究生. 主要研究方向为区块链安全.

Luo Xiapu, born in 1977. PhD, professor. His main research interest includes blockchain security.
罗夏朴, 1977 年生. 博士, 教授. 主要研究方向为区块链安全.

Chen Ting, born in 1987. PhD, professor. His research interests include blockchain, smart contract, and software security.
陈　厅, 1987 年生. 博士, 教授. 主要研究方向为区块链、智能合约、软件安全.

Luo Ruijie, born in 1998. Master. His main research interest includes blockchain security.
罗瑞杰, 1998 年生. 硕士. 主要研究方向为区块链安全.



Wang Bingsen, born in 1998. Master. His main research interest includes blockchain security.
王炳森, 1998 年生. 硕士. 主要研究方向为区块链安全.

Qiao Ao, born in 1999. Master candidate. His main research interest includes blockchain security.
乔　翱, 1999 年生. 硕士研究生. 主要研究方向为区块链安全.