软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(10):4681−4704 [doi: 10.13328/j.cnki.jos.006664]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于录制重放的区块链交易执行追溯方法*

陈 胜[1], 方明哲[2], 蒋步云[1], 李春晓[2], 左 春[2,3], 李玉成[2,4], 梁 赓[2]

[1](北京连琪科技有限公司, 北京 100086)

[2](中国科学院 软件研究所, 北京 100190)

[3](中科软科技股份有限公司, 北京 100190)

[4](贵阳信息技术研究院, 贵州 贵阳 550081)

通信作者: 梁赓, E-mail: lianggeng@iscas.ac.cn

摘 要: 区块链上运行的智能合约具有一经部署难以修改、调用执行需经过共识等特点, 现有的需要修改智能合约代码或打断其执行过程的调试方法难以直接应用到智能合约上. 由于智能合约的运行过程由区块链交易顺序执行过程组成, 实现对区块链交易执行过程的追溯是提升智能合约可调试性的一个有效途径. 对区块链交易执行过程进行追溯主要目标是找出一条已经出块的区块链交易是如何得到当前的执行结果的. 区块链交易的执行依赖于区块链内部状态, 且该状态取决于之前区块链交易的执行结果, 因此存在着传递性依赖. 区块链交易的依赖性和区块链所提供的执行环境的特点给区块链交易执行追溯带来了挑战. 区块链交易执行追溯面临的挑战主要有 3 方面, 即如何从智能合约部署的生产环境中获取足够追溯的信息、如何获取区块链交易之间的依赖关系, 以及如何保证追溯结果与实际在线执行过程一致. 提出了一种基于录制重放的区块链交易执行追溯方法, 在合约容器中建立录制重放机制, 无需修改合约代码即可支持交易执行中对状态读写操作的录制, 并且不会打断智能合约运行; 提出了基于状态读写的交易依赖分析算法, 支持对存在依赖关系的前序交易进行按需回溯; 此外, 设计了录制读写操作记录的验证机制, 确保重放的执行过程与真实执行过程之间的一致性可被验证. 所提出的方法能够追溯区块链交易调用智能合约的执行过程, 可用于智能合约调试, 并且当智能合约异常造成损失时可用于举证. 在实验中对比了将录制的读写操作记录存储于链上和存储于链下之间的性能差异, 通过案例研究展示了所提方法在追溯区块链交易执行方面的有效性和优点.

关键词: 区块链; 交易追溯; 智能合约

中图法分类号: TP311

中文引用格式: 陈胜, 方明哲, 蒋步云, 李春晓, 左春, 李玉成, 梁赓. 基于录制重放的区块链交易执行追溯方法. 软件学报, 2023, 34(10): 4681–4704. http://www.jos.org.cn/1000-9825/6664.htm

英文引用格式: Chen S, Fang MZ, Jiang BY, Li CX, Zuo C, Li YC, Liang G. Tracing Method for Blockchain Transaction Execution Based on Recoding and Replay. Ruan Jian Xue Bao/Journal of Software, 2023, 34(10): 4681–4704 (in Chinese). http://www.jos.org.cn/1000-9825/6664.htm

Tracing Method for Blockchain Transaction Execution Based on Recoding and Replay

CHEN Sheng[1], FANG Ming-Zhe[2], JIANG Bu-Yun[1], LI Chun-Xiao[2], ZUO Chun[2,3], LI Yu-Cheng[2,4], LIANG Geng[2]

[1](Linkel Technology Co. Ltd., Beijing 100086, China)

[2](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[3](Sinosoft Co. Ltd., Beijing 100190, China)

[4](Guiyang Academy of Information Technology, Guiyang 550081, China)

* 基金项目: 国家重点研发计划 (2020YFC1523203)

收稿时间: 2021-08-09; 修改时间: 2021-12-21; 采用时间: 2022-02-22; jos 在线出版时间: 2022-09-20

CNKI 网络首发时间: 2023-01-05

Abstract: Smart contracts running on the blockchain can hardly be modified after deployment, and their call and execution rely on a consensus procedure. Consequently, existing debugging methods that require the modification of the smart contract code or the interruption of execution cannot be directly applied to smart contracts. Since the running of a smart contract is composed of ordered execution of blockchain transactions, tracing the execution of the transactions is an effective approach to render the smart contract more debuggable. The major goal of tracing blockchain transaction execution is to unveil how a blockchain transaction produces such a result in execution. The execution of a blockchain transaction relies on the internal state of the blockchain, and this state is determined by the execution results of previous transactions, which results in transitive dependencies. Such dependencies and the characteristics of the execution environment the blockchain provides bring challenges to tracing. The tracing of blockchain transaction execution is mainly faced with three challenges: how to obtain enough information for tracing from the production environment in which the smart contract is deployed, how to obtain the dependencies among the blockchain transactions, and how to ensure the consistency between the result of tracing and the real execution online. This study proposes a tracing method for blockchain transaction execution based on recording and replay. By building a recording and replay mechanism in the contract container, the proposed method enables the recording of state reading and writing operations during transaction execution without modifying the contract code and interrupting the running of the smart contract. A transaction dependency analysis method based on state reading and writing is proposed to support the retracing of previous transactions linked by dependencies on demand. Moreover, a verification mechanism for reading and writing operation recording is designed to ensure that the consistency between the replaying execution and the real online execution can be verified. The tracing method can trace the execution of the blockchain transaction that calls the smart contract, which can be used in debugging of smart contracts. When loss is caused by the failure of smart contracts, the tracing result can be used as evidence. Experiments are conducted for a performance comparison between storing recorded reading and writing operations on chain and off chain. The advantages and effectiveness of the proposed method in tracing blockchain transaction execution are revealed by a case study.

Key words: blockchain; transaction tracing; smart contract

区块链为智能合约[1]的正确执行提供了分布式执行结果共识[2]、链式数据结构可追溯防篡改等特性的保障[3], 使得智能合约日益流行起来. 支持图灵完备语言的智能合约能够实现复杂的业务逻辑, 能够支撑记账与支付[4]、身份认证与访问控制[5]、数据共享[6,7]等现代信息系统的核心业务功能在区块链上的实现, 推动了区块链技术在供应链管理、电子政务、数字金融等领域开始得到广泛的应用[8].

与传统业务应用的请求-响应模式不同, 基于区块链的应用系统将部分业务逻辑写入部署在区块链上的智能合约, 由用户直接或间接地提交经过用户数字签名的区块链交易调用和触发智能合约中的方法, 由参与共识的区块链节点各自执行并对执行结果进行共识以确保正确性, 输出信息记录到具有链式结构、能够防篡改、冗余地存储于各个区块链节点的区块数据中, 链下的业务模块通过同步得到区块数据并据此执行业务动作. 在上述过程中, 区块链交易调用智能合约的执行的正确性对业务应用至关重要, 虽然区块链提供了共识机制确保执行结果是多方达成共识的可信结果, 然而智能合约代码本身可能存在缺陷, 共识得到的结果只不过是“正确地”得到了一个错误的结果. 无论是因为主观恶意还是因为过失和疏忽, 执行结果一旦与预期不符, 都会引起不良后果. 例如研究者分析以太坊公有链上以诈骗用户财产为目的蜜罐合约, 发现仅 690 个蜜罐智能合约造成了高达 90 000 美元的直接损失[9]. 相比于自组织的公有链, 联盟链在应用中常支撑一系列企业、政务等业务系统, 智能合约缺陷导致区块链交易执行结果错误进而造成业务系统失灵, 能够造成更为直接的经济损失和恶劣影响. 联盟链设施的参与方对于联盟链设施的正常运行负有责任与义务, 因此更加迫切地需要技术手段能够发现问题、消除隐患、挽回损失.

目前, 对于已经部署于生产环境运行的智能合约进行调试和分析的研究较少, 出现错误后难以快速定位问题的源头. 智能合约的活跃周期可划分为 3 个主要阶段, 即智能合约设计与开发阶段、智能合约编译与部署阶段、智能合约运行阶段. 现有的关于减少智能合约缺陷的研究主要集中在前两个阶段. 其中, 针对智能合约设计与开发阶段, 主要从智能合约的描述语言、建模工具和实现方法等方面[10−13]确保智能合约代码能够更为精准地表示承载的业务活动, 虽然能够有效减少设计与开发过程中引入的错误, 但是这类方法仅能起到辅助作用, 无法排除恶意引入的智能合约漏洞, 同时也无法消除编译与部署过程中引入的缺陷. 针对智能合约编译与部署阶段, 研究者引入软件验证与测试技术[14−23], 预先设定需要满足的一系列性质或检查项, 在智能合约部署执行前对其逻辑进行验证以确保其满足安全性质, 或通过检查排除其中的缺陷. 这类工具方法能够被集成到区块链平台上确保智能合约均得到验证或检查, 其局限性在于依赖先验知识, 难以应对新型漏洞, 并且通常较为耗时. 针对智能合约运行阶段的研

究[24−27]主要以确保智能合约得到安全可信的执行为目标, 相比智能合约本身更侧重执行环境的安全性. 然而, 在智能合约运行阶段, 能够对区块链交易执行过程进行分析和调试, 对于定位问题的源头有着更为直接的帮助. 此外, 还原执行过程还能够作为利益相关方协商解决缺陷带来损失的依据.

针对生产环境运行的软件进行分析和调试在软件调试领域本身就是一个具有挑战性的问题, 区块链系统特殊的运行环境更为分析和调试智能合约增加了难度. 其中, 部署到区块链上的智能合约难以直接进行修改, 因此需要修改代码的调试方式行不通. 此外, 通过断点等中断执行过程的方式进行调试, 则会影响区块链共识过程, 可能影响执行结果. 记录日志[28,29]同样是一种掌握软件运行情况的手段, 然而需要智能合约开发者事先自行决定输出哪些日志并编写日志记录语句, 目前仍离不开定制化开发, 需要更新已经部署的智能合约. 即使通过定制化开发为区块链节点增加输出日志的能力, 由于缺少可验证的机制, 无法证明给出的日志与生产环境中实际在线执行过程是一致的. 此外, 现有方法难反映区块链交易之间的依赖关系. 智能合约的单次被调用执行所依赖的当前状态是之前的执行结果, 为了分析当前执行情况往往需要回溯之前的调用执行, 然而现有的反向执行调试技术[30,31]主要针对的是更为底层、粒度更细的命令和语句.

针对智能合约运行阶段区块链交易执行难以进行分析和调试的不足, 以联盟链为主要研究对象, 研究了区块链交易执行追溯问题, 即找出区块链交易的执行过程如何产生当前的执行结果. 具体地, 本研究的目标是解决追溯区块链交易执行 3 方面挑战: 如何从智能合约部署的生产环境中获取足够的信息来进行追溯、如何获取区块链交易之间的依赖关系、如何保证追溯结果与实际在线执行过程的一致性. 针对这些挑战, 本文采用了在线记录执行信息与离线分析相结合的思路, 提出了一种基于录制重放的区块链交易执行追溯方法. 本文主要贡献如下.

- 针对从智能合约部署的生产环境中获取用于追溯的信息的问题, 建立区块链交易执行录制重放机制, 通过增强合约容器, 录制状态读写操作用于之后离线分析和重放, 无需修改智能合约代码或中断执行过程, 录制得到的状态读写操作记录可按需进行离线重放.
- 针对获取区块链交易之间的依赖关系问题, 提出基于世界状态读写的交易依赖分析方法, 以录制得到的区块链交易读写操作记录为基础, 提取交易读写键集合, 从而能够利用读写键集合识别交易之间的依赖关系, 可沿依赖关系按需逐条回溯所依赖的前序交易进行重放和分析, 实现区块链交易执行追溯.
- 针对保证追溯结果与实际在线执行过程的一致性的问题, 建立读写操作记录可验证机制, 实现在链上存储和链下存储两种情况下能够验证读写操作记录是否正确未经篡改.

在比较实验中考察了将录制的读写操作记录链上存储和链下存储之间的性能差异, 通过案例研究展示了本方法的有效性和优点. 本文提出的方法能够在无需对现有智能合约进行修改和更新、无需要求新开发的智能合约增加日志输出代码的情况下, 利用合约容器录制得到用于追溯的信息, 并能够根据区块链交易执行中不同区块链交易读写键重叠的情况分析区块链交易之间的依赖关系实现对区块链交易的按需回溯重放与分析, 并提供了追溯结果与实际在想执行过程一致性的保障机制. 本方法能够为智能合约缺陷的定位和解决提供帮助, 并且能为联盟链场景中利益相关方通过协商挽回损失提供依据.

本文第 1 节对区块链交易执行追溯问题进行了分析. 第 2 节介绍了所提出的基于录制重放的区块链交易执行追溯方法. 第 3 节介绍了实现所提出方法的技术细节. 第 4 节通过实验分析与案例研究展示了所提出方法性能方面的影响以及其有效性. 第 5 节介绍了相关工作. 第 6 节总结全文, 并对下一步工作做出展望.

1 区块链交易执行追溯问题

1.1 区块链交易执行过程

与区块链平台进行互操作的主要方式之一就是提交附有数字签名的区块链交易到区块链平台接口, 以调用智能合约, 如图 1 所示. 合约容器为智能合约提供运行环境, 所执行的交易的内容作为具体调用方法的输入, 在智能合约的方法中通常会涉及对区块链状态进行读写操作, 成功执行的交易会与其他交易一起打包为区块 (block), 记录到链上区块数据中, 这一动作被简称为出块. 其中, 负责存储区块链状态的模块可被看作是位于区块链内部的键

值 (key-value) 型数据库, 被称作世界状态 (world state). 不同交易的执行会对相同的键进行读写, 后执行的交易读取得到的值为先执行的交易写入的值, 存在依赖关系. 由于先执行的交易对于更早执行的交易同样存在着依赖关系, 区块链交易之间的依赖具有传递性, 为分析和研究区块链交易执行过程带来了挑战.

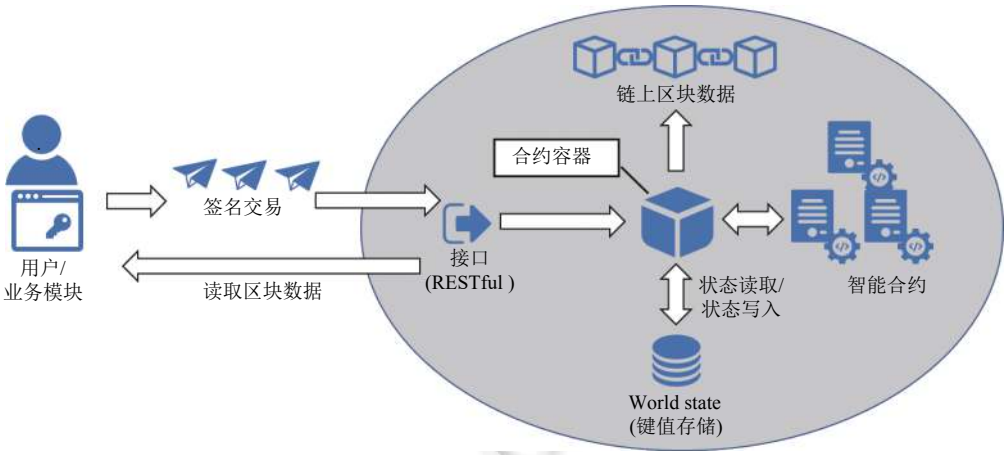


图 1 区块链交易执行过程

下面以转账智能合约为例介绍一个具体的交易执行过程. 转账智能合约在区块链世界状态中存储了用户的账户余额, 其中用户 A 和用户 B 的余额对应的键 (也被称作地址) 分别为 a32be7 和 7e1a9d, 其中的值为以元为单位、数据类型为浮点数的用户余额. 用户 A 通过提交交易给用户 B 转账 10 元的执行过程为:

(1) 进行一系列检查与验证, 验证交易的数字签名及用户 A 的身份, 检查用户 A 与用户 B 的键是否存在等.

(2) 读取键 a32be7 对应的值 getState (a32be7), 判断转账后满足账户余额不为负的约束 getState(a32be7)–10> 0, 如不满足则停止, 如满足则执行步骤 (3).

(3) 执行转账操作, 具体包括 setState(a32be7, getState(a32be7)–10) 和 setState(7e1a9d, getState(7e1a9d)+10).

其中, getState(key) 为针对键 key 当前值的读操作, setState(key, value) 为将 value 写入键 key 的值的写操作. 可以看出, 本条交易能否成功执行取决于键 a32be7 对应的值, 而该值取决于之前涉及这个键的写操作. 假设两个键当前取值均为 100, 则交易能够成功执行, 执行完交易后 a32be7 取值 90, 而 7e1a9d 取值 110.

区块链共识机制确保交易执行后, 整个区块链在世界状态和输出区块数据方面的一致性. 对于安全性较强的支持拜占庭容错的共识算法, 通常由多个区块链组网节点维护各自的合约容器, 各自执行经过共识的一个区块链交易有序集合, 并对执行结果也进行验证和共识. 为了应对真实生产环境中的系统和通信故障, 目前常见的区块链平台的共识算法通常设有等待超时机制.

1.2 区块链交易执行追溯挑战

区块链交易执行追溯的目标是找出区块链交易的执行过程如何产生当前的执行结果. 正如第 1.1 节所述, 区块链交易执行依赖于区块链交易的内容所提供的输入和区块链内部的世界状态, 并且不同交易对于相同的世界状态键的读写使得区块链交易之间存在着传递性依赖. 如图 2 的右半部分所示, 一方面, 区块链交易的内容所提供的输入可通过访问公开的区块数据获得, 而区块链内部的世界状态难以直接获取, 即使区块链平台提供公开的底层世界状态访问接口, 也只能取得最新的状态 (目前仅有以太坊的存档节点实现了世界状态历史快照的支持, 由于硬件条件要求很高, 实际部署并提供服务的存档节点非常稀少). 另一方面, 区块链交易之间依赖关系无法通过交易本身进行判断, 只有区块链交易执行中才能够检查其中是否存在相同键的读写. 此外, 在实际中如果希望能够利用区块链交易执行追溯结果作为证据, 用于利益相关方协商以挽回损失, 难以证明包含了单机离线计算步骤的追溯得到的结果与区块链上经过共识的在线执行过程的一致. 综上所述, 解决区块链交易执行追溯问题需要面对以下 3 方面挑战.

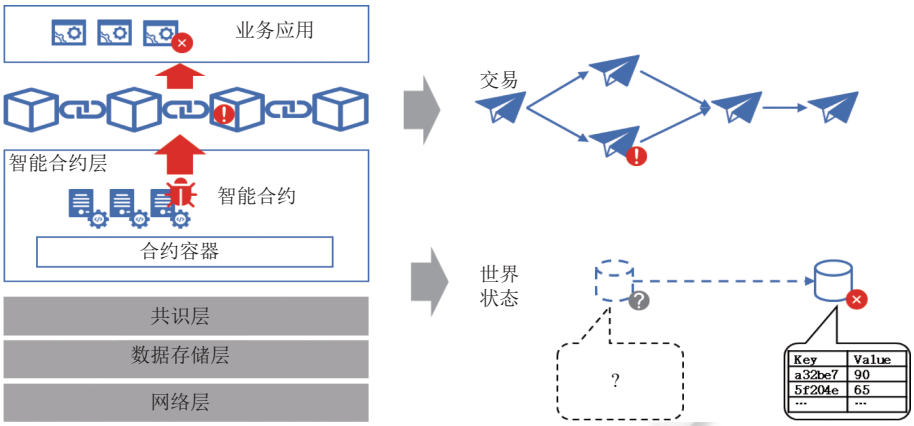


图 2 区块链交易执行追溯场景

(1) 如何从智能合约部署的生产环境中获取足够的信息来进行追溯?

对于运行在生产环境中的软件, 如何从中获取用于追踪数据变化的调试信息本身就是一个难题. 区块链在安全性和性能方面的要求使得现有技术难以直接应用到区块链交易执行追溯. 由于区块链交易执行结果需要经过分布式共识, 现有的基于断点调试的方法会中断执行过程, 引起共识步骤的超时, 可能造成本应成功的共识结果变为失败, 进而影响区块链正常运行. 图 3 给出了直观的基于离线执行的追溯方法. 一种方法是离线执行交易, 为了复现在线执行过程需要从区块数据加载从创始区块至今的全部交易并执行, 然而这种方法仅在理论上可行, 在区块链上庞大的交易数量面前行不通, 并且难以向前回溯已经执行过的交易. 另一种方法是通过设置检查点 (checkpoint), 定期转储世界状态得到其快照, 可在快照基础上执行其后的交易而无需从创始区块开始执行. 然而世界状态中包含的键值对数量巨大, 转储的存储成本很高, 还面临转储频率与快照存储成本之间的取舍. 除了上述方法, 一种更加可行的思路是将在线记录信息与离线分析重放结合[32], 即在生产环境中对在线执行情况进行记录, 将计算量较大的观察和分析放到离线进行. 采用这种方法仍需涉及区块链执行过程在线记录与离线重放机制, 需要考虑对区块链运行的额外消耗.

(2) 如何获取区块链交易之间的依赖关系?

无论是从业务应用的角度, 还是从数据访存的角度, 区块链交易之间存在着传递性的依赖关系. 对于一条区块链交易, 其执行结果不仅取决于交易本身的内容, 还取决于执行区块链交易所依赖的世界状态. 具体而言, 区块链交易的依赖是由对世界状态中相同键的读写产生的. 如图 4 所示, 其中箭头代表了对世界状态中键值的读写操作, 其中交易 3 对键 a32be7 的值的读取操作依赖于交易 2 写入的结果, 而交易 2 对键 5f204e 的读取操作依赖于交易 1. 对于区块链交易执行过程的追溯, 如果仅仅止步于复现追溯对象的执行过程, 在实际使用中难以发现智能合约运行故障产生的根本所在, 缺少实际应用价值. 如果能够获取区块链交易之间的依赖关系, 能够向前回溯所依赖的交易, 能够找出造成当前区块链交易执行结果的真正原因. 目前, 对于程序执行回溯的研究主要集中在程序反向调试领域, 大部分现有的方法和工具针对的是较为底层的指令或语句[30,33], 尚未有专门针对区块链交易执行的研究.

(3) 如何保证追溯结果与实际在线执行过程的一致?

区块链本身提供了共识机制保证区块链交易能够被正确执行, 然而对于区块链交易执行追溯, 在离线分析阶段通过离线执行去重现交易缺少同样的机制保证执行过程与真实在线执行过程之间的一致性. 缺少这样的机制则追溯结果的正确性无法保证, 并且存在可篡改的风险, 即使得到追溯结果也无法将其用于解决智能合约故障. 需要建立一种可验证的机制保证追溯结果与在线执行过程的一致性.

本文研究的区块链交易执行追溯的最终目的是为发现和解决区块链智能合约缺陷提供帮助, 主要关注的是智能合约缺陷, 不考虑智能合约运行环境异常 (如网络异常) 的情况. 如图 2 的左半部分所示, 在区块链系统中, 当智能合约存在缺陷, 执行结果错误, 对世界状态的修改以及写入区块数据的内容也是错误的, 进而影响业务应用. 由

于存在共识机制, 在恰当的参数设定下, 即使合约容器和智能合约层之下的 3 层存在缺陷, 错误的结果也很难进入到区块数据, 要么交易未能出块被丢弃, 要么大概率采用其他正常区块链节点执行的结果, 与此相比智能合约的缺陷危害更大更为直接.

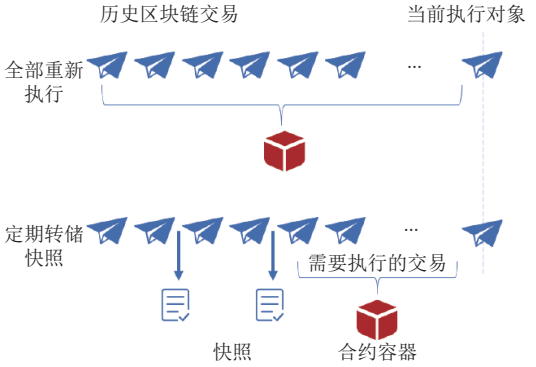


图 3 区块链交易离线执行——全部执行和定期转储快照

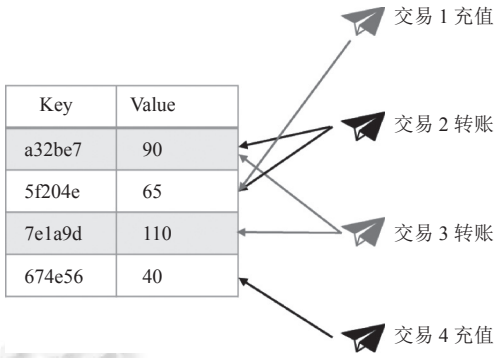


图 4 区块链交易之间的依赖

2 基于录制重放的区块链交易执行追溯方法

2.1 方法概述

针对上述 3 方面挑战, 本文提出的基于录制重放的区块链交易执行追溯方法的总体思想是采用在线录制与离线分析结合[32,34]的思路, 通过写-读依赖对交易执行进行追溯, 能够录制区块链交易执行过程并进行回溯和按需重放. 当区块链交易执行因智能合约缺陷而出现错误, 执行后造成区块链上世界状态与预期不符, 即可利用本方法定位问题的根源, 重放并观察交易的执行, 并且能够回溯到所依赖的前序交易进行重放.

本文提出的方法主要分为 3 个部分.

(1) 针对区块链交易执行的记录, 通过增强合约容器, 记录读写操作用于之后离线分析和重放, 无需修改智能合约代码或中断执行过程. 在重放阶段, 取得区块链交易的读写操作记录和区块交易数据, 即可利用合约容器按需离线重放, 以区块链交易为粒度观察和分析区块链交易执行情况和执行过程对世界状态的读写操作, 实现对区块链交易的追溯. 读写操作记录支持写入区块数据或发送到日志采集系统, 采用后者能降低区块数据的额外存储消耗.

(2) 针对建立区块链交易之间的依赖关系, 以录制得到的区块链交易读写操作记录为基础, 提取交易读写键集合, 从而能够利用读写键集合识别交易之间的依赖关系, 能够在重放中回溯到所依赖的前序交易进行观察和分析. 此外, 结合数字签名与智能合约开发、部署、调用的流程, 能够建立区块链交易执行结果与其开发者、业务活动参与者之间的关联, 形成完整证据链, 作为挽回损失和解决纠纷的依据.

(3) 针对建立追溯结果与实际在线执行过程一致性, 如果选择将读写操作记录写入区块数据, 由于这些记录与区块链交易一样经过了区块链节点的共识, 能够保证读写操作记录的正确性和不被篡改. 如果采用将这些记录通过日志采集系统发送到链下存储的策略, 则在区块数据中存储录制的读写操作记录的默克尔根, 确保得到的记录是可验证的.

所提出的基于录制重放的区块链交易执行追溯方法的完整执行步骤如图 5 所示, 可分为: ① 基础设施与智能合约部署、② 运行与录制、③ 追溯与重放这 3 大环节. 各个环节的具体执行步骤在图中的下方展开列出. 其中, 构建区块链应用的通用步骤以非加粗字体表示, 本方法中包含的针对性改造所涉及的步骤则以加粗字体表示.

第 1 个环节是在业务活动开始前, 联盟链的运营联盟进行区块链基础设施部署和组网, 开发者和普通用户注册, 开发者将业务应用所依赖的合约部署到区块链上. 本方法要求在各个参与方维护的参与共识的节点上配置改造后的附带录制功能的合约容器以代替原有合约容器, 如采用链下存储策略则还需联盟链运营联盟协商后部署一

个 (或多个) 日志采集系统用于收集录制的读写操作记录以备将来进行缺陷修复、挽回损失等使用. 对于联盟链中不参与共识的节点, 如不需要重放功能, 则对其是否部署改造后的合约容器没有强制要求, 可兼容原有容器. 联盟链运营联盟部署持续运行的依赖分析服务, 在用户请求读写操作记录的时候可以返回“脱水”后的读写操作记录, 即过滤掉不存在依赖关系的交易对应的读写操作记录, 并且提供已经提取出的区块链交易之间的依赖关系.

图 5 基于录制重放的区块链交易执行追溯方法执行步骤

第 2 个环节是业务系统和区块链运行中, 用户账户通过构建区块链交易调用智能合约、触发业务逻辑的执行, 联盟链管理员能够对智能合约进行各种管理操作. 在这一阶段开发者可以将更新的智能合约部署到区块链上, 在联盟链管理员、业务应用开发者共同操作下将原有请求对象和数据源指向切换到新版本智能合约, 这也是能够修复智能合约缺陷的基础. 在这一环节中, 用户通过交易调用智能合约会触发合约容器的录制功能, 如采用链下存储策略录制的读写操作记录会由日志系统收集, 如采用链上存储策略则会随交易一起被打包记录到区块链上.

第 3 个环节是出现问题后进行追溯和重放. 追溯和重放等操作无需中断区块链平台的运行, 参与业务的利益相关方 (个人用户、应用服务提供商、开发者等) 经联盟链授予访问权限, 从链上或日志系统取得疑似存在缺陷的智能合约被调用产生的读写操作记录, 之后可以利用链上数据验证记录是否真实有效, 进行依赖关系判定与提取, 利用合约容器进行交易的回溯和按需离线重放, 并以此作为依据进行进一步处理, 例如由开发者进行缺陷修复, 由联盟链代表和利益相关方协商挽回损失的方案. 其中, 也可利用联盟链运营方提供的依赖分析服务, 提供得到的“脱水”后的仅包含关联交易的读写操作记录. 为了避免更大的损失, 发现问题后可以马上由联盟链管理员暂时冻结智能合约, 待经过缺陷修复后才重新启用.

图 6 给出了不同角色的参与者执行动作的流程. 所提出的方法主要适用于带有背书机制的联盟链, 此外在不考虑利用背书机制确保追溯结果与实际执行过程一致、读写操作记录可验证的情况下, 也可将该方法扩展应用到公有链上. 图 7 展示了有无背书机制的区别. 其中图 7(a) 展示了联盟链背书机制, 提议者将区块链交易和自己的执行结果 (对于世界状态的修改) 发送到区块链网络, 参与共识的节点收到广播的区块链交会执行交易对结果进行验证, 并通过数字签名来表示对执行结果的认可, 即对其背书. 其中, 部分区块链平台 (如 RepChain[35]) 的背书对象是以排序、打包好的若干交易组成的区块为单位, 而部分区块链平台 (如 Hyperledger Fabric) 则以单个交易为单位进行背书而将排序打包步骤放到后面. 一旦背书比例超过某一预设阈值则该交易即准备被打包出块, 其他节点无需执行合约而是在检查数字签名和背书符合规则后, 直接进行世界状态的修改和本地区块数据的写入. 在这一场景下, 共识节点的背书代替了节点自己执行和验证. 其中, 在联盟链下共识节点由联盟的核心组织或个人维护, 因此通常假设其不会主动作恶. 本方法中第 3 部分将读写操作记录本身或其默克尔根记录到区块数据中, 因此可

验证性与联盟链区块数据一样依赖于背书机制. 具体地, 所提方法除了适用于本文实验所采用的 RepChain 外, 还适用于同样适用于 Hyperledger Fabric 和 FISCO BCOS 等区块链平台.

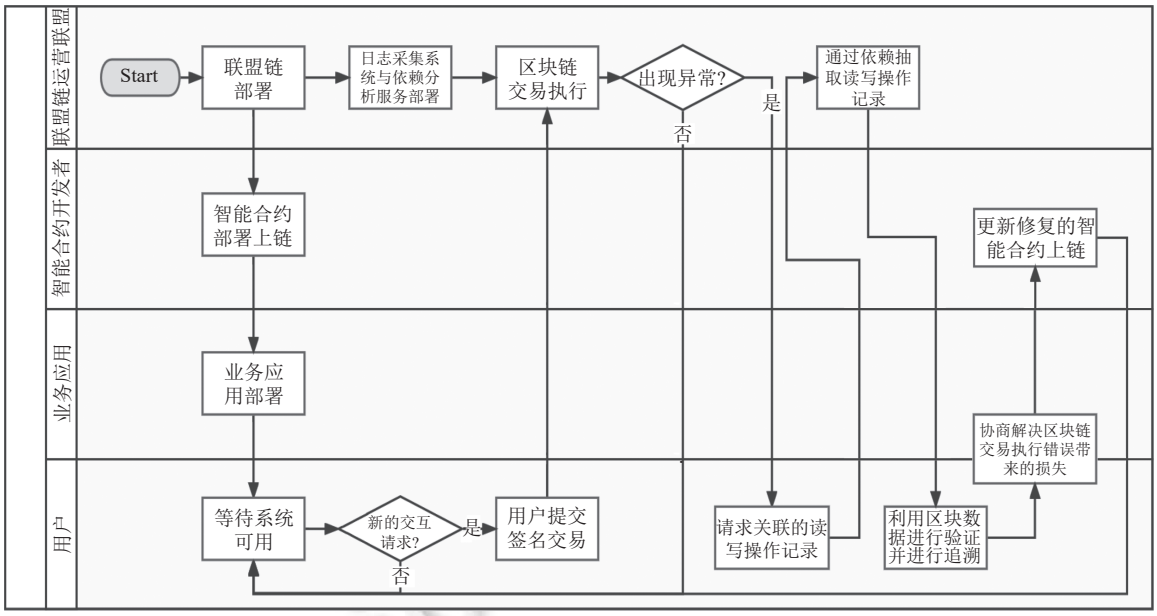
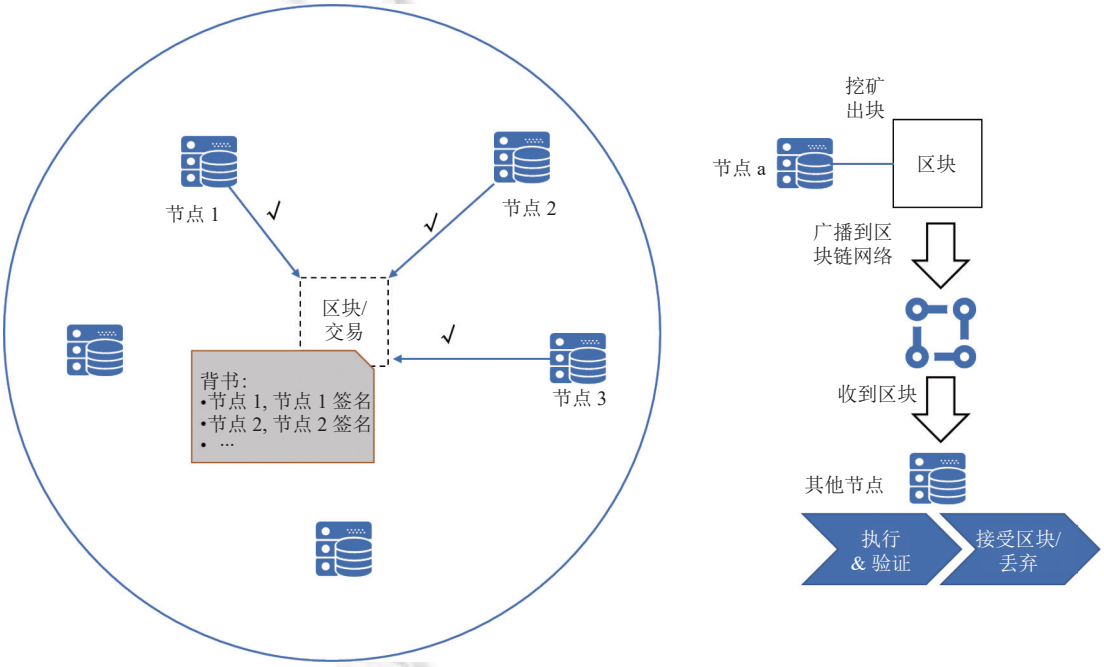


图 6　不同角色的参与者动作流程



(a) 有背书机制　　(b) 无背书机制

图 7　联盟链共识的背书机制

图 7(b) 给出的公有链执行交易的一般过程, 一个区块链节点通过挖矿即运行工作量证明 (proof of work) 算法获得出块的权利后, 自己执行交易后得到世界状态的最新默克尔根, 将交易、默克尔根打包到区块中, 附上挖矿成

功的证明和数字签名进行广播, 其他节点则均需要执行交易. 在这样的场景下, 应用本方法第 3 部分是无意义的, 但是其他两部分内容依然适用于公有链, 可用于智能合约的分析和调试. 此外, 在联盟链中, 本方法第 3 部分的一个作用是可以作为证据, 由联盟链代表和利益相关各方协商以挽回损失, 而公有链是自组织的没有类似的机制, 因此即使能够应用第 3 部分, 也无法发挥类似联盟链中的作用.

2.2 区块链交易执行录制重放机制

为了从区块链交易执行过程中获取足以用于进行追溯的信息, 设计了区块链交易执行录制重放机制. 在交易执行阶段进行录制, 一方面需要在执行中记录足以用于重放区块链交易执行过程的数据, 另一方面需要提取足以建立区块链交易之间依赖关系的数据. 对于前者, 根据第 1.1 节中所述区块链交易执行过程, 区块链交易执行依赖的数据主要包含两方面信息, 即区块链交易提供的输入和区块链内部的世界状态. 其中, 区块链交易提供的输入主要是作为智能合约方法的调用实参, 保存在公开的区块数据中, 可公开获取, 无需额外进行记录. 至于世界状态, 通常难以直接获取, 且即使能够通过底层数据访问接口访问世界状态, 取得的也仅仅是最新的世界状态, 而难以取得历史某一个时间点的快照. 为此, 正如第 2.1 节所述, 本研究采用了录制的思路, 即对区块链交易执行过程中的世界状态读写进行记录. 对于后者, 正如第 1.2 节对于区块链交易之间依赖的分析, 其主要表现为对世界状态中相同键的读写, 利用这一表现建立区块链交易之间依赖的前提同样是对世界状态读写进行记录. 综上所述, 实现区块链交易执行录制重放机制的一个关键是能够对世界状态读写进行记录.

区块链交易执行中进行世界状态读写通常是调用内部的应用程序接口实现 (API) 的, 提供这一接口的上下文由合约容器维护. 受到智能合约执行必须是确定性的这一原则约束[36], 无论是区块链平台内置的智能合约还是第三方开发的智能合约, 改变世界状态均需要通过上述途径. 通过对合约容器进行增强和改造, 即可实现对世界状态读写操作的记录, 且不影响区块链系统的运行. 特别地, 由于改造的是合约容器, 即使在进行世界状态读写操作前对键值进行变换, 记录的是变换后最终读写操作, 并且在录制和重放时会运行相同的智能合约程序逻辑, 不会影响追溯效果.

如后文图 8 所示, 图 8(a) 展示了原始的合约容器, 区块链交易的执行过程中直接调用原始内部 API. 图 8(b) 为经过改造和增强的合约容器, 从调用和实现两个角度保证增强的合约容器能够在不影响原有系统功能和运行的前提下实现读写操作记录. 首先, 从调用角度, 合约容器内部 API 中包括了 getState(key) 和 setState(key, value) 等方法供合约开发者调用, 采用依赖倒置原则让具体的智能合约程序逻辑不再依赖具体实现, 而是依赖于上述方法组成的抽象接口. 无论是原有内部 API 提供者还是能够记录世界状态读写操作的可录制 API 提供者均实现这一抽象接口. 由合约容器维护运行时采用哪一个提供者, 提供者对于智能合约的开发者和调用者来说是透明的. 其次, 从实现角度, 可录制 API 提供者内部通过命令模式实现了对于世界状态读写操作的记录, 在内部维护原始内部 API 实例和一个执行队列, 将该实例提供的状态读写方法封装成输出记录信息的命令, 而可录制 API 提供者实现的读写方法只需要将封装好的命令压入执行队列, 读写操作即被调用并产生读写操作记录. 在第 3 节中给出了基于命令模式的参考实现的介绍和 UML 模型.

在抽象接口的基础上, 通过录制得到的读写操作记录能够用于离线重放交易的执行过程. 具体地, 在离线重放中, 由一个同样实现了抽象接口的可重放 API 提供者提供重放版世界状态读写方法, 所提供的世界状态读写方法无需访问世界状态, 而是加载录制好的读写操作记录进行执行. 所建立的录制重放机制满足了进行区块链执行追溯的第 1 个要求, 即记录单一交易执行所依赖的数据. 所录制的读写操作记录与真实线上执行过程之间的一致性由第 2.4 节保证. 对于第 2 个要求, 即建立区块链交易之间的关系, 则是利用读写的世界状态的键之间的重叠关系为基础, 设计了第 2.3 节的方法.

2.3 基于世界状态读写的交易依赖分析方法

进行区块链交易执行追溯, 需建立区块链交易之间的依赖关系, 并基于依赖关系向前回溯与之关联的历史交易, 能够观察影响当前执行结果的各个环节. 为了能够建立交易之间的依赖关系, 首先要能够判定两个区块链交易存在直接依赖. 其次, 需要从全部历史交易中提取所有依赖关系. 最后, 依赖关系要能被用于对关联的历史交易按需进行逐条回溯重放.

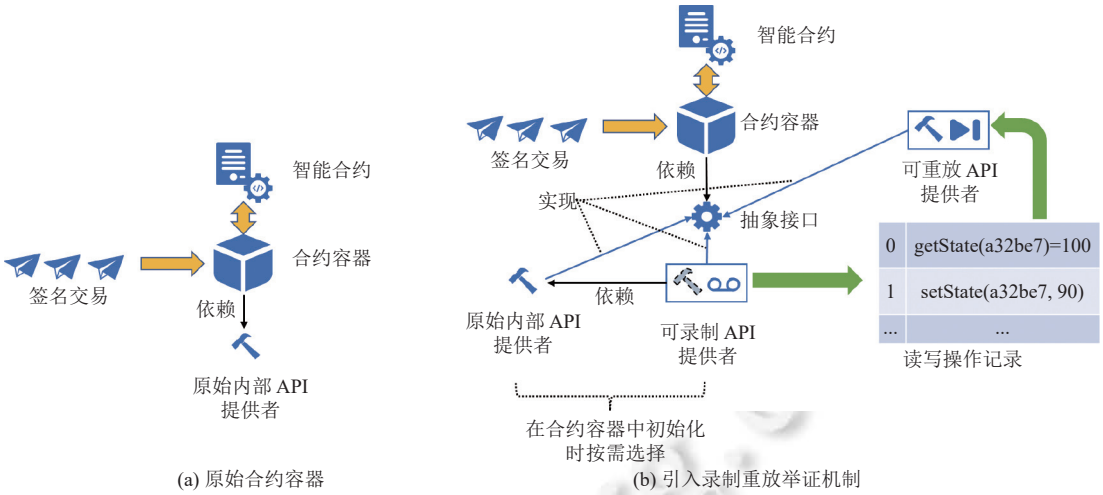


图 8 区块链交易执行录制重放机制

(1) 直接依赖关系判定

判定两个区块链交易存在直接依赖, 可以简单地通过区块链交易执行时读写的世界状态的键的重叠情况进行判断. 对于两个不同的交易中的两个读写操作, 当它们读写对象为相同的键, 按照二者先后执行顺序, 可能存在 4 种情况: 读-读、读-写、写-读和写-写. 其中, 连续两个读取操作之间不构成依赖关系; 对于连续两个写操作, 后续操作直接覆盖原有值, 同样不存在依赖; 对于读-写操作, 如果属于同一交易则可能是依据读取结果计算后赋值, 然而二者分处不同交易, 后者直接进行写操作对前者并无依赖; 对于写-读关系, 读操作所处交易后续程序逻辑依赖于之前写入的值. 因此, 本文采用与 Light 技术[37]相同的思路, 通过写-读关系判断依赖. 为了便于找出包含所依赖交易的区块, 在分析中同时提取出区块间的依赖关系.

将交易 τ 的读操作键的集合记作 R_τ, 写操作集合键的集合记作 W_τ.

定义 1. 交易间依赖. 对于两条不同的交易 τ_i 和 τ_j, 将后者对前者存在依赖当且仅当 $W_{\tau_i} \cap R_{\tau_j} \neq \varnothing$, 记作 $\tau_i \to \tau_j$.

定义 2. 区块间依赖. 对于两个不同的区块 B_x 和区块 B_y, 后者对前者存在依赖, 记作 $B_x \to B_y$, 当且仅当存在 $(\tau_i, \tau_j) \in B_x \times B_y$ 使得 $\tau_i \to \tau_j$.

(2) 依赖关系提取算法

通过对区块链交易的一次遍历, 提取全部区块链交易间依赖以及区块间依赖, 如算法 1 所示. 将区块链交易间依赖组成的图记作 $G_{\text{transaction}}$, 将区块间依赖组成的图记作 G_{block}, 其中添加有向边的操作记作 addEdge, 为幂等操作. 其中, 交易列表 T 中的交易严格按照交易执行顺序进行排序, 在遍历过程中访问的区块链交易的区块高度也是呈单调递增的. 通过第 2.2 节得到的读写操作记录被用于获取各条交易对于世界状态读写涉及的键的集合, 为简便起见, 分别将一条交易 τ 执行时写入的键的集合和读取的键的集合的获取函数记作 $W(\tau)$ 和 $R(\tau)$. 算法中用到了一个名为 WDict 的散列表, 其中以交易执行中出现的世界状态的键为该散列表的键, 其中的值为遍历至今对于该世界状态的键最新的一次写操作对应的交易.

算法 1. 区块链交易依赖关系提取.

Input: T;

Output: $G_{\text{transaction}}$, G_{block}.

1. 初始化 $G_{\text{transaction}}$ 和 G_{block} 为邻接表形式图表示结构
2. WDict = new HashTable()
3. **for** j←1, 2, …, $|T|$ **do** // 顺序处理交易

```
4.      τ←T[j] // 取出一条交易
5.      b←T[j].blockHeight // 获取其区块高度
6.      WKeySet←W(τ)
7.      RKeySet←R(τ)
8.      for RKey in RKeySet do // 遍历集合
9.          if WDict.hasKey(RKey) do // 判断 Map 中是否存在
10.           G_transaction .addEdge(WDict[RKey], τ)
11.           G_block .addEdge (WDict[RKey].blockHeight, b)
12.     end for
13.     for WKey in WKeySet do
14.           WDict[WKey]←τ
15.     end for
16. end for
17. return G_transaction , G_block
```

(3) 交易执行回溯

在建立区块链交易之间的依赖关系后, 给定一条交易, 可以利用区块链交易关系组成的图向上回溯, 按需重放路径上的交易. 其中, 区块链交易关系组成的图结构为有向无环图, 沿着依赖关系回溯可能会出现分叉, 但是不会出现回环, 因此不会出现需要重新分析某些交易的情况, 这一性质保障了本方法的有效性. 下面对此进行分析.

在区块链上, 区块交易的执行顺序经过共识, 在执行和写入区块时具有统一的先后顺序. 由此, 给定全部区块交易的集合 T, 其中交易执行顺序关系记作 $<$, 满足如下性质.

① 反自反性. $\forall \tau \in T, \tau \not< \tau$.

② 禁对称性. $\forall \tau_i, \tau_j \in T, \tau_i < \tau_j \Rightarrow \tau_j \not< \tau_i$.

③ 传递性. $\forall \tau_i, \tau_j, \tau_k \in T, (\tau_i < \tau_j) \wedge (\tau_j < \tau_k) \Rightarrow \tau_i < \tau_k$.

根据严格偏序关系的定义, $<$ 为 T 上的严格偏序关系. 区块链交易的执行只能依赖于当前交易提供的输入和区块链状态, 因此有如下公理.

公理 1. $\forall \tau_i, \tau_j \in T, \tau_i \to \tau_j \Rightarrow \tau_i < t_j$, 即对于任意一条交易, 它所依赖的交易的执行顺序在该交易之前.

定理 1. 以交易间的依赖为有向边, 交易为节点, 组成的图结构是有向无环图 (DAG).

证明: 采用反证法, 假设存在环路径 $P: \tau_j \to \tau_i \to \tau_k \ldots \to \tau_j$; 根据公理 1 可得到路径 P 上由依赖连接的交易两两之间的关系, 即 $\tau_a < \tau_b (\tau_a, \tau_b \in P \wedge \tau_a \to \tau_b)$; 根据性质③, 将两两之间的关系推广到整个路径, 可推出 $\tau_j < \tau_j$, 与性质①矛盾, 证毕.

后文图 9 给出了基于区块链交易依赖关系进行交易执行回溯的流程. 针对建立区块链交易之间的依赖关系, 以录制得到的区块链交易读写操作记录为基础, 提取交易读写键集合, 从而能够利用读写键集合识别交易之间的依赖关系. 结合数字签名与智能合约开发、部署、调用的流程, 能够建立区块链交易执行结果与其开发者、部署者、业务活动参与者之间的关联, 结合智能合约绑定的描述其功能的法律文本, 形成完整证据链, 作为挽回损失和解决纠纷的依据.

2.4 读写操作记录可验证机制

确保建立追溯结果与实际在线执行过程一致的关键在于确保读写操作记录真实有效, 给定一组读写操作记录, 需要能够验证它是否与在线执行过程一致. 此处提供两种方法.

(1) 将读写操作记录写入区块数据

区块数据是经过共识产生的, 冗余地存储在每个区块链节点上, 是公开、可验证的. 确保读写操作记录可验证的一种方法就是将其也写入到对应交易相同的区块中. 在向区块数据存储读写操作记录的时候, 同样以区块链交易为单位进行组织, 确保一个区块中的交易和读写操作记录之间对应, 方便进行查看、分析和处理. 将读写操作记

录写入区块数据的缺点是会给区块链运行带来一定的额外存储和计算开销, 优点是无需额外的系统或模块负责收集读写操作记录, 也无需建立额外的验证机制来对读写操作记录进行验证, 这些都可以由区块数据本身的特性来保证.

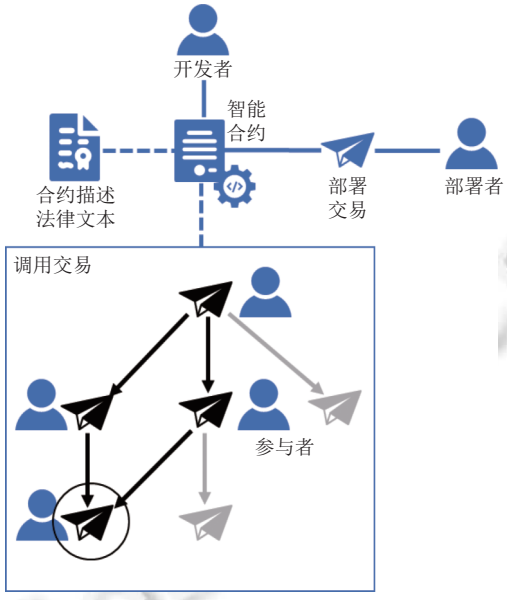


图 9 区块链交易执行回溯

(2) 基于默克尔树的验证方法

将读写操作记录写入区块数据会带来一定的额外开销, 保存读写操作记录的另一种方式是不将读写操作记录写入区块链, 而是发送给外部的日志收集系统或者模块, 然而存储于链下的数据与区块链交易的在线执行过程之间缺少强制关联. 采用默克尔树技术能够建立二者之间的可验证绑定关系. 如图 10 所示, 在将区块交易打包出块之前, 对这些交易对应的读写操作记录构建默克尔树, 将默克尔根作为一个属性记录到区块数据中. 为了减少构建默克尔树带来的额外消耗, 同时考虑到对区块链交易执行进行分析时数据组织的粒度通常也是以交易为单位的, 因此在构建默克尔树时选择以一条交易对应的一系列读写操作而不是单独一条读写操作打包进行哈希作为叶子节点. 按执行先后顺序对区块中每条交易的读写操作记录进行上述操作形成了叶子节点, 之后将叶子节点成对拼接后哈希, 得到上一层节点. 为了避免一层节点为奇数时带来的安全隐患 (CVE-2012-2459), 在计算上层节点时先将奇数层节点末尾节点复制作为自己计算上层哈希的配对.

当进行区块链交易执行追溯, 给定区块链交易的读写操作记录, 构建默克尔树, 将默克尔根与区块数据中记录的默克尔根进行比对, 即可验证区块交易读写操作记录的真实性, 避免伪造数据造成的追溯结果与真实过程不一致.

3 实 现

本方法实现的核心是通过增强合约容器以实现区块链交易在线执行过程的录制. 本方法的实现选用了开源区块链平台 RepChain[35]作为基础. RepChain 本身以 Scala 语言实现, 采用了响应式消息框架 Akka, 在实现录制重放机制时, 无论是选择将读写操作记录存储到区块数据中, 还是选择将其发送给外部日志存储系统并在区块数据中存储默克尔根, 均可通过 Akka 框架将任务分配给单独计算单元执行, 从并发程序开发的角度这种实现方式具有较低的额外消耗.

RepChain 中的智能合约开发所需要的世界状态读写等内部 API 方法被封装在名为 Shim 的类中, 供智能合约的开发者调用, 具体由 ContractContext (图 11 左上角) 作为上下文对象的属性提供给开发者. 对于世界状态的访问, 由于其采用了键值型的存储结构, 可以简单地概括为 getState(key) 和 setState(key, value). 键和值的具体结构根据智能合约的需要进行设计, 可能存在多种类型, 在转账操作智能合约中键代表着用户账户唯一标识而值为余额,

对于账户管理智能合约则通常以用户身份唯一标识为键、以包含用户证书的用户属性序列化数据为值. 因此, 为便于实现重放, 在实现录制读写操作记录时对不同类型的值均进行统一的序列化存储.

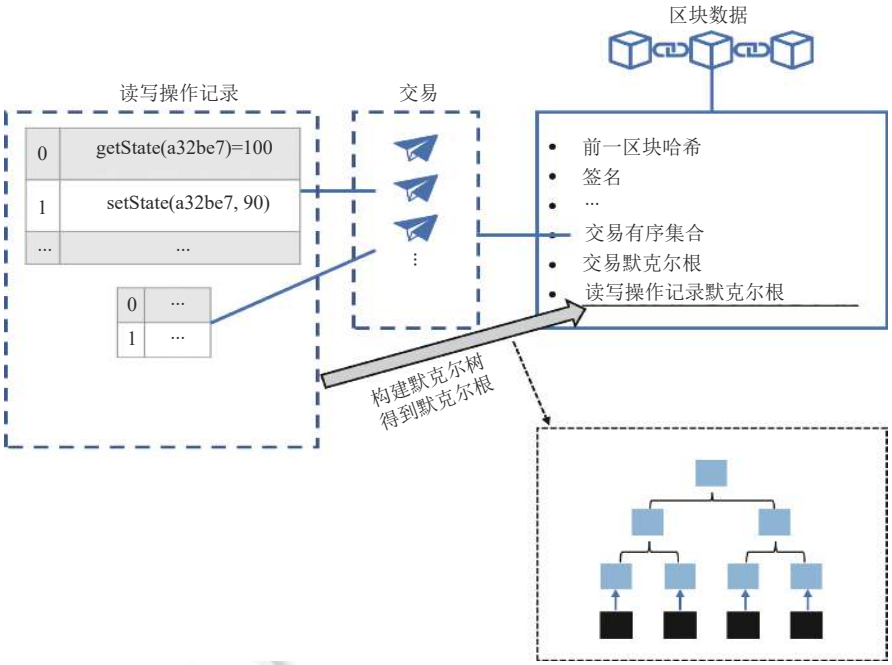


图 10 基于默克尔根的读写操作记录验证

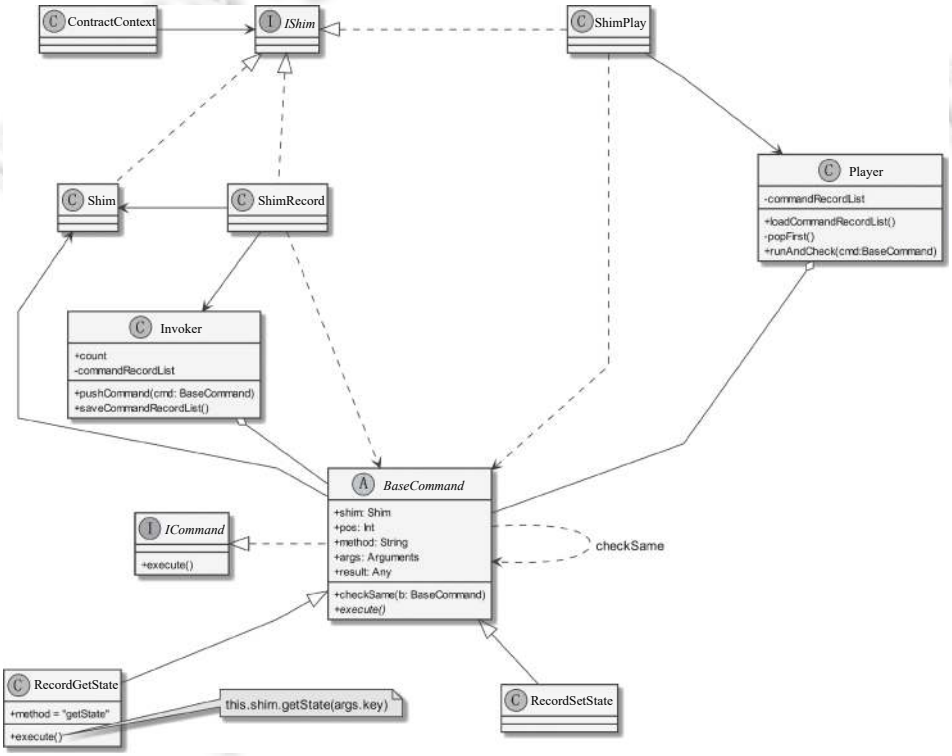


图 11 基于命令模式的录制重放机制参考实现

图 11 给出了为实现录制重放机制对智能合约容器进行的改造的一种基于命令模式的参考实现. 这种实现方式的优点在于无需修改原始内部 API 提供者 Shim, 可录制 API 提供者完全兼容原有接口, 可以通过配置选择是否启用录制重放功能. 此外, 当需要新增其他需要录制的世界状态读写方法, 可在不破坏已有结构的前提下进行扩展.

具体地, ContractContext 不再依赖于原始提供者 Shim 而是依赖于抽象接口 *IShim*. 可录制 API 提供者 ShimRecord 和可重放 API 提供者 ShimPlay 分别实现了该接口, 前者维护了 Shim 对象的实例去执行底层真正的读写操作, 后者则只需要加载和重放读写操作记录而无需进行底层读写. 对于底层读写操作, 均封装成一个个命令, 供 ShimRecord 和 ShimPlay 调用. 二者维护的调用者 (Invoker 和 Player) 的区别决定了实际执行底层读写操作还是从重放读写操作记录. 例如, ShimRecord 中的 getState 方法的实现是以合约承载的调用参数作为参数, 实例化命令封装类 RecordGetState, 并将实例交给 Invoker 执行并压入读写操作记录队列的过程, ShimPlay 中 getState 的实现则是由 Player 从读写操作记录队列中取出当前的 getState 操作在线执行时录制的结果进行重放, 其形式同样为 RecordGetState 的实例. 当出现新的世界状态读写方法 (*IShim* 中新增方法), 只需要随之在 ShimRecord 和 ShimPlay 中实现新的方法, 并以抽象命令类为基础实现新的命令封装类即可, 无需对录制重放机制本身进行修改.

在数据存储方面, 对于直接写入区块数据的方式, 调用平台内部区块数据写入接口, 让读写操作记录与区块链交易一起经过分布式执行和验证, 最后出块, 成为可验证、防篡改的区块数据的一部分. 对于链下存储的方案, 则设计了如图 12 所示的方案, 以消息队列中间件 RabbitMQ 为基础, 通过一个日志采集系统收集区块链交易执行时产生的读写操作记录, 并对读写操作记录进行聚合. 其中, 由于部分共识算法中区块链交易执行存在预执行和正式执行等执行次数超过一次的情况, 为确保数据采集的健壮性通常会采集来自多个节点的数据, 在聚合中也要对这种情况进行处理, 形成与直接写入区块数据方式相同的记录. 根据共识算法的特点, 采用两种聚合策略, 即:

(1) 面向 CFRD 的聚合策略: 对于一条交易仅当收集到来自超过 1/2 的节点的一致日志时才将其聚合和处理并写入数据存储.

(2) 面向 Raft 的聚合策略: 直接对单一出块节点发送的读写操作记录进行处理和记录.

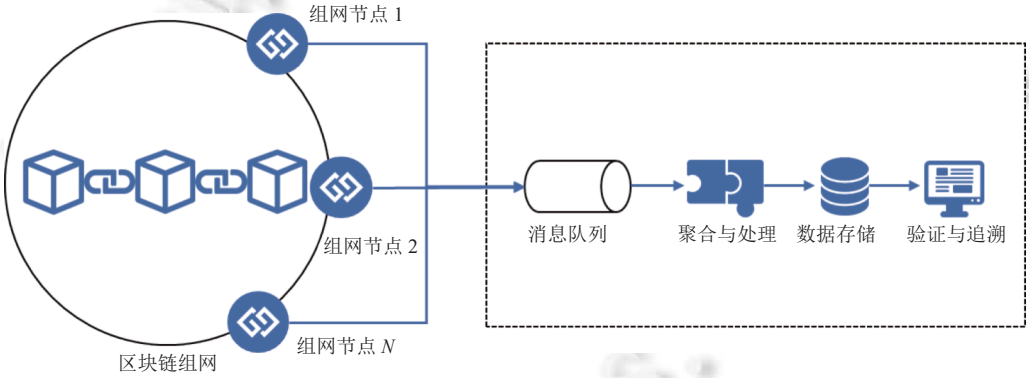


图 12 读写操作记录链下存储的实现

采用上述两种聚合策略的原因在于根据采用的共识算法的不同, 智能合约代码执行次数和执行者也不同, 产生的记录的副本数也不同, 可利用多副本记录实现拜占庭容错. 由于 CFRD 是拜占庭容错共识算法, 为实现拜占庭容错各节点会分别执行合约代码, 对于一条记录会生成来自不同节点的副本, 因此所设计的面向 CFRD 的聚合策略在聚合中利用来自多节点的副本防止采集中的拜占庭错误. Raft 本身不支持拜占庭容错, 选举单一节点执行合约代码和出块, 仅有一个节点产生读写操作记录, 因此在聚合中仅作简单的处理和记录, 也无法支持拜占庭容错.

与写入区块数据的链上存储相比, 链下存储对区块链系统的性能影响更小, 在通过 Akka 框架实现异步 I/O 的情况下这种性能影响几乎可以忽略. 但是无法提供区块链系统所提供的冗余存储于多个节点不易丢失的特性. 此外, 链下存储可直接将合约执行日志连续、完整地保存到同一个文件或数据表, 无需像链上存储那样从不同区块取出后进行合并.

4 实验分析与案例研究

在本节中, 对于将录制的读写操作记录存储于链上和链下两种情况下的性能进行了实验分析. 为了简便, 在后文中将录制的读写操作记录存储于链上的方案简写为 OnCR (on chain record), 将后者存储于链下的方案简写为 OffCR (off chain record). 在二者之间, OnCR 从原理上具有更大的额外消耗, 因此首先分析了 OnCR 实现与区块链平台原有实现之间的性能差异. 之后, 分析对比了 OnCR 和 OffCR 的功能和性能. 最后, 通过案例研究展示了所提出方法用于追溯区块链交易执行的有效性.

4.1 性能分析

主要从两个角度考察 OnCR 实现相比原有实现 (在实验结果中记作 Original) 在性能上的差异: (1) 在高速提交交易情况下, 满负荷 (即交易到达速度大于等于处理速度) 的系统对于交易的处理能力; (2) 在低速提交交易情况下交易出块的延迟情况. 在上述两种情况下进行实验分析的目的在于对 OnCR 进行较为全面的检验, 一方面不应该在系统满负荷的情况下因录制功能的加入而出现严重的瓶颈, 导致系统更容易在高速提交交易场景下失效; 另一方面, 对在低速提交交易情况下, 不应该造成严重的交易出块延迟增加的情况.

实验采用如下实验设置.

- 共识算法: 在 RepChain 平台自主研发的无协商随机抽签算法 (consultation-free random draw algorithm, CFRD) 和 Raft 算法上进行实验.
- 服务器设备: 双路 Intel Xeon E5-2683 v4 @ 2.10 GHz 处理器, 128 GB 内存, Windows Server 2008 R2 SP1 操作系统.
- 部署方式: 采用单服务器多节点部署, 部署 5 个组网节点, 且 5 个节点全部为共识节点.
- 其他系统参数: 设置区块大小上限为 81 个交易, 对于 Raft 协议出块者轮换条件为达到 100 个块即进行轮换.
- 实验数据: 分别采用交易数量为 10 万、20 万、40 万条的 3 组数据进行实验, 数据内容为随机转账操作.
- 交易提交: 在与服务器通过千兆网络连接的 PC 上进行交易提交, PC 拥有 Intel Core i7-7700 @ 3.60 GHz 处理器和 16 GB 内存, 通过多线程自动化提交程序提交交易, 可按需设置并发数和等待延迟.

为了兼顾前文提到的两个考察的角度, 使系统能够在高速提交交易情况下达到满负荷, 在组网规模和区块大小等参数选择上采用了较为保守的设置.

(1) 高速提交交易情况下

对于高速提交交易情况下区块链系统的处理能力, 可采用全部完成时间和实际平均 TPS (transactions per second) 两个指标进行分析. 全部完成时间指的是从提交第 1 条交易到最后一条交易出块所耗费的时间. 实际平均 TPS 则统计了平均每秒钟有多少交易被系统处理完毕后出块. 这里的 TPS 计算方法与压力测试中的 TPS 计算方法略有不同, 后者仅考虑所发出的请求得到响应. 为了避免区块链系统在初始阶段与临近结束阶段正在处理的交易量较小导致数据不准确, 在测量实际平均 TPS 的时候, 依惯例取中间 1/3 的交易.

图 13 展示了高速提交交易情况下全部完成时间情况, 分成了 10 万交易、20 万交易和 40 万交易 3 组. 每一组内分别展示了 Raft 和 CFRD 两种共识算法与原有实现和 OnCR 实现产生的 4 种组合情况的对比. 可以看出无论是在 Raft 上, 还是在 CFRD 上, OnCR 实现的完成时间均略长于原有实现. 其中, 在 40 万交易上, CFRD 上的 OnCR 实现比原有实现多耗费了 22.8% 的完成时间. 这一数据从实际经验来看尚处于可接受范围内, 未出现数量级的变化.

图 14 展示了高速提交交易情况下中间 1/3 交易的实际平均 TPS 的情况, 同样分成了不同交易规模的 3 组进行对比. 从实际平均 TPS 的角度能够得到与图 13 相似的结论, OnCR 实现对于实际平均 TPS 有一定的影响, 但是并未造成 TPS 的大幅缩水. 其中差异较大的是 40 万交易上采用 CFRD 算法的情况, OnCR 实现比原有实现在实际平均 TPS 上低 18.6%.

通过进一步分析和观察发现, 在高速提交交易情况下, 大部分区块中的交易数都达到了预设的上限 (81 条). 在采用随机转账数据的测试条件下, 对于 OnCR 实现产生的区块, 其平均单个区块字节大小比原有实现大 69%,

可以看出区块增大与区块链系统的性能降低并非呈线性关系. OnCR 带来的数据存储空间额外消耗受采用的序列化方式影响, 在实际应用中可以引入压缩通过牺牲一定的性能来减少区块增大的字节数.

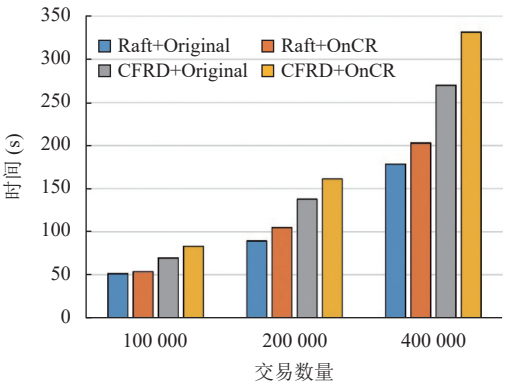


图 13 高速提交交易情况下全部完成时间对比

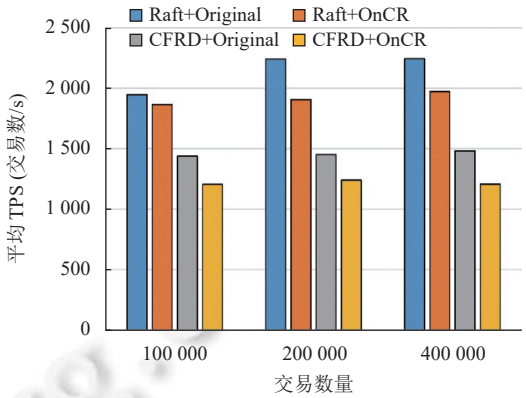


图 14 高速提交交易情况下实际平均 TPS 对比

(2) 低速提交交易情况下

由于在高速提交交易情况下, 在系统满负载情况下, 后提交的交易会被放入缓存队列中等待处理, 因此难以知晓一条交易记录所耗费的实际处理时间. 在低速提交交易的情况下, 可以利用出块延迟分析区块链系统处理每条交易所耗费的时间. 出块延迟指的是从提交交易到交易被打包出块所耗费的时间, 在低速提交交易的情况下交易无需进入缓存排队, 因此出块延迟约等于处理每条交易所耗费的时间.

图 15 和图 16 分别展示了低速提交交易情况下, 基于 CFRD 和 Raft 的原有实现与 OnCR 实现的出块延迟分布情况. 其中 (a), (b), (c) 3 组分别为 10 万交易、20 万交易和 40 万交易的情况, 每组字母后带有序号 1 的为原有实现, 带有序号 2 的为 OnCR 实现. 为便于观察, 纵轴采用了对数坐标.

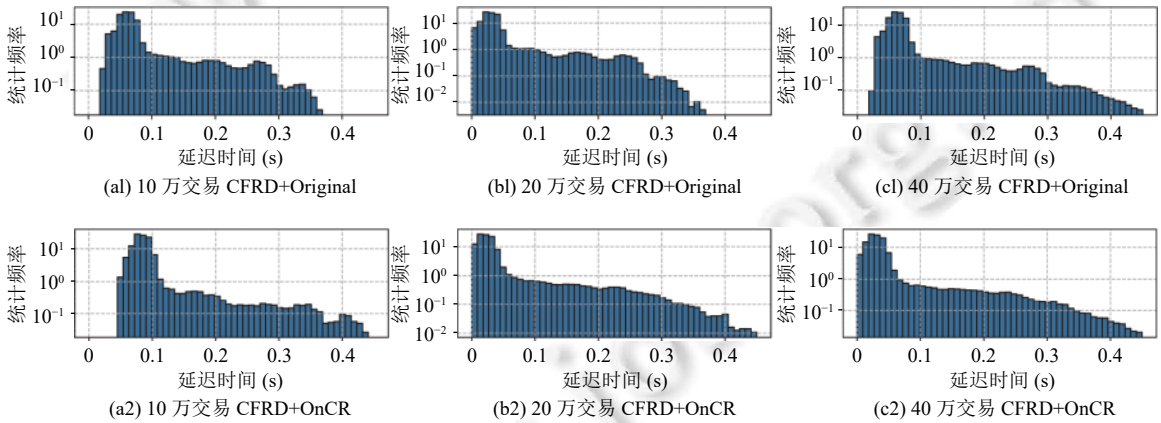


图 15 原有实现与 OnCR 实现在低速提交交易情况下出块延迟分布情况对比, 采用 CFRD 共识算法

根据图 15, 从整体上看, 原有实现与 OnCR 实现二者分布的形状相似、区间接近, OnCR 实现的出块延迟的尾巴要更长一些, 即与原有实现相比存在较多的出块延迟较高的情况. 注意, 由于图中是纵轴采用对数坐标得到的结果, 这是由于在常规坐标下尾巴带来的差异是难以观察的. 根据图 16, 由于 Raft 算法本身在共识效率方面具有优势, 基于 Raft 的原有实现的出块延迟分布较为紧凑. OnCR 实现的出块延迟则更加分散, 尾巴更长一些, 这一现象与 CFRD 上相似.

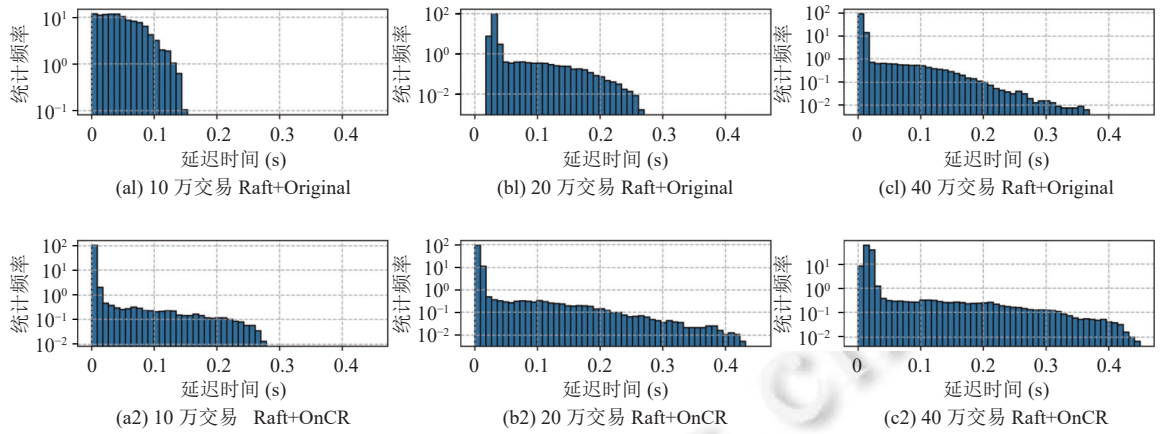


图 16 原有实现与 OnCR 实现在低速提交交易情况下出块延迟分布情况对比, 采用 Raft 共识算法

总的来看, 如果采用链上存储的方式存储读写操作记录会带来额外的性能消耗, 增大了区块的体积, 降低区块链系统在满负荷情况下的处理能力, 增加高速提交的批量交易的处理时间, 增加每条交易的处理时间. 不过这种性能消耗仍处于可接受的范围内, 并未造成处理能力下降程度发生数量级的变化.

4.2 方法对比

对比了 OnCR 实现与 OffCR 实现之间功能和性能上的差异. 实验设置如下.

系统概况: 系统架构如图 12 所示, 将日志采集系统置于与服务器通过千兆网络连接的 PC 上, PC 拥有 Intel Core i7-7700 @ 3.60 GHz 处理器和 16 GB 内存, 通过 RabbitMQ 收集智能合约容器发送的世界状态读写操作记录, 之后对来自不同节点的日志记录以交易 ID 和节点名称为线索进行聚合和处理, 形成支持分析和追溯的统一记录.

消息传输机制: 采用无确认机制的简单发送模式并且不启用事务模式, 这是因为启用事务或确认机制会阻塞智能合约的执行过程, 造成共识过程效率低下, 且易导致在预出块、背书等阶段超时.

(1) 功能对比

OnCR 实现与 OffCR 实现之间的功能对比如表 1 所示. OffCR 实现虽然能够对给定的读写操作记录进行验证, 但是并不意味着总是能准确无误地得到一份读写操作记录. 在单点故障方面, OnCR 实现读写操作记录会与交易数据一起共识出块, 因此自然地能够抵御单点故障; OffCR 也可以基于分布式容错的日志系统抵御单点故障, 但是需要额外的协调、通信的开销, 代价较高. 在保证读写操作记录与交易数据对应方面, OffCR 收到记录与对应的交易成功出块之间没有必然联系, 如果聚合中增加检查和等待对应的交易成功出块的功能模块, 则也会产生额外的内存和通信开销. 最后, OffCR 实现中的分布式日志采集系统本身并不支持能够与链上存储相类似的可靠数据存储. 通过对比可以看出, OffCR 虽然性能上具有天然的优势, 但是实现表 1 中列出的功能特性需要付出一些额外的代价.

表 1 功能对比

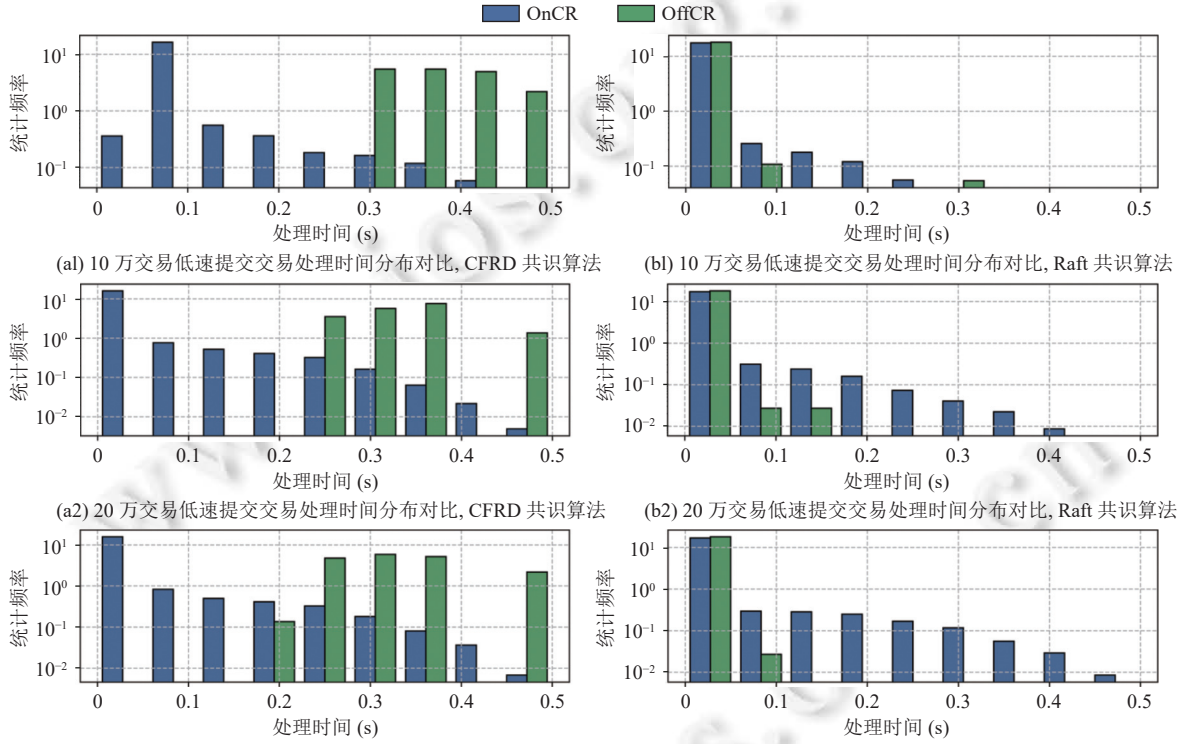
特性	OffCR		OnCR	
	面向CFRD的聚合策略	面向Raft的聚合策略	采用CFRD共识算法	采用Raft共识算法
抵御单点故障	代价较高	代价较高	√	√
保证日志与交易数据对应	代价较高	代价较高	√	√
日志采集的拜占庭容错	√(存在额外开销)	×	√	×
可靠数据存储	×	×	√	√

(2) 处理时间对比

OnCR 实现和 OffCR 实现的差异主要体现在从交易提交到形成一致日志所耗费的时间, 因此对比了低速提交

交易情况下两种方法的处理时间, 以分析性能上的差异. 对于 OnCR 实现, 处理时间仍采用从提交交易到读写操作记录出块所耗费的时间; 对于 OffCR 实现则采用从提交交易到聚合处理后形成一致的读写操作记录所耗费的时间. 注意, 对于 OffCR 实现, 在记录被发送到消息队列后, 后续处理过程均在区块链系统外进行, 不对区块链系统造成额外性能消耗, 因此 OffCR 实现的处理时间组成的主要部分为链下处理时间. 在 CFRD 和 Raft 两种共识算法上分别针对 10 万、20 万、40 万交易进行了实验, 针对两种共识算法分别采用了前文中提到的聚合策略.

实验结果如图 17 所示. 为便于观察, 纵轴采用了对数坐标. 针对 CFRD 上的实验结果 (图 17(a1)–图 17(a3)), 可以看出 OnCR 实现的处理时间分布情况要优于 OffCR 实现的情况. 初步分析其原因在于相比直接随着交易在本地共识出块, 读写操作记录通过消息队列再经过聚合处理耗费较多时间. 进一步分析发现, OffCR 从读写操作记录发送到聚合处理完成平均占用了整个处理时间的约 84%, 这一阶段并不会对区块链系统带来性能消耗. 在 Raft 上的实验结果 (图 17(b1)–图 17(b3)) 则呈现出相反的结果, 即采用 OffCR 实现的处理时间优于 OnCR. 其原因在于在 Raft 共识算法中, 仅选出的单一节点出块, 聚合处理过程较为简单.



(a1) 10 万交易低速提交交易处理时间分布对比, CFRD 共识算法　(b1) 10 万交易低速提交交易处理时间分布对比, Raft 共识算法

(a2) 20 万交易低速提交交易处理时间分布对比, CFRD 共识算法　(b2) 20 万交易低速提交交易处理时间分布对比, Raft 共识算法

(a3) 40 万交易低速提交交易处理时间分布对比, CFRD 共识算法　(b3) 40 万交易低速提交交易处理时间分布对比, Raft 共识算法

图 17　OnCR 和 OffCR 的处理时间对比

综合功能和处理时间的对比以及第 4.1 节的实验结果可以看出, 在不考虑拜占庭容错、可靠数据存储以及抵御单点故障等特性的情况下, OffCR 在性能方面更有优势, 在安全性和健壮性要求较高的场景中 OnCR 实现是更合理的选择, 其带来的性能额外消耗处于可接受范围.

4.3　案例研究

为了展示本方法的有效性, 设计了一个委托代销分账的案例, 分析如何对委托代销分账智能合约进行存证. 委托代销是一种常见的业务模式, 委托方 (通常为商品的供应商) 和受托方 (零售商) 签订代销分账协议, 委托受托方向终端客户销售商品. 在受托方向最终客户出售商品之前, 委托方拥有对商品的控制权, 受托方无需承担对商品无条件付款的业务. 图 18 展示了一种实施委托代销的具体方式的示意图. 可以看出, 在这种模式下销售者向零售商

下单, 之后发货和库存管理由供应商负责, 零售商无需承担库存积压的风险, 销售所得由支付平台直接分账. 这种零售商负责销售、供应商直接发货模式非常适合销售保质期相对较短、不宜中转运输和储藏的农产品, 以及快速更新换代的科技产品.

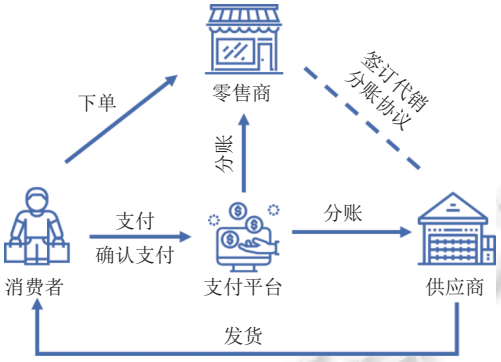


图 18 代销分账示意图

目前实施委托代销分账需要依赖于第三方支付平台, 受限于第三方支付平台的功能限制, 而通过传统技术自建代销分账系统则存在各方面的风险. 通过区块链对于数据的可追溯、不可篡改的存储能力以及灵活可靠的智能合约, 能够解决这一问题. 下面给出一个委托代销智能合约的描述 (图 19), 为便于说明, 本案例只考虑针对单一商品的一个零售商和一个供应商的场景, 该场景可以很容易地扩展到多商品、多方的场景.

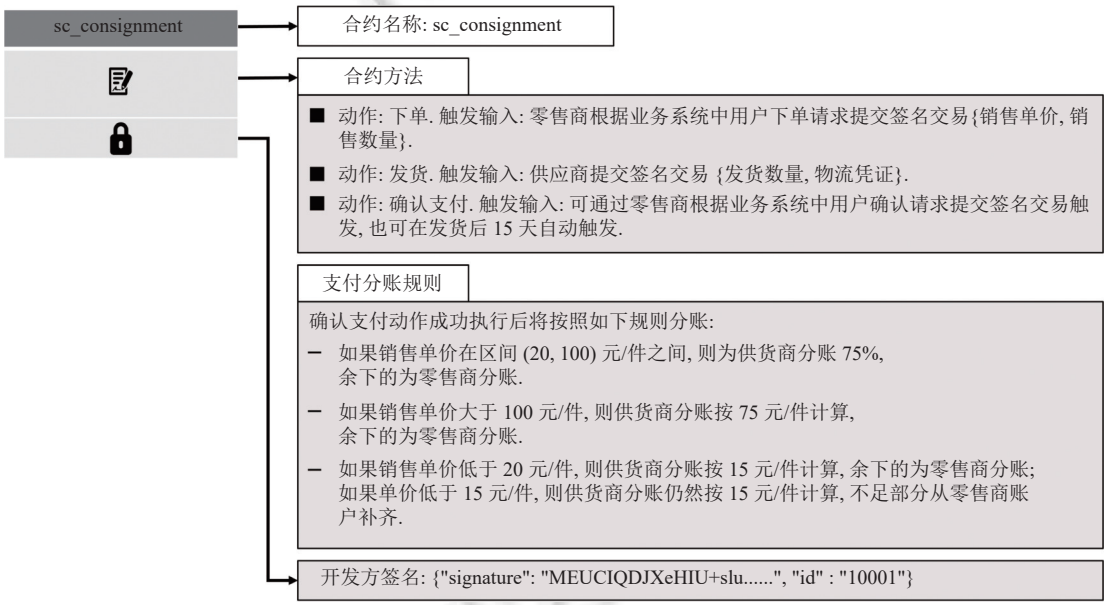


图 19 代销分账智能合约

假设该智能合约的开发方程序员在开发上述合约时粗心大意, 在单价低于 20 元/件, 忘记从零售商账户扣除用于补齐供货商分账不足 15 元/件的部分的款项. 如果在交易中存在单价低于 20 元/件的情况, 则必然会造成零售商最终账户余额错乱. 为了便于说明, 给出一个更为具体的场景.

零售商为了吸引更多消费者, 在秒杀活动中将 100 件商品以 1 元/件的价格售出; 后续以 50 元/件的促销价格售出 80 件商品; 之后由于供货紧张, 商品价格升至 110 元/件, 售出商品 30 件. 在这一过程中, 如果合约能够正确

执行, 分账情况为:

- 供货商分账 $15\times100+75\%\times50\times80+75\times30=6750$ 元.
- 零售商分账 $(1-15)\times100+25\%\times50\times80+(110-75)\times30=650$ 元.
- 总的销售所得 $100+50\times80+110\times30=7400$ 元, 与供货商和零售商分账之和相等.

然而, 由于开发的合约代码存在 bug, 则会出现:

- 供货商分账 $15\times100+75\%\times50\times80+75\times30=6750$ 元.
- 零售商分账 $25\%\times50\times80+(110-75)\times30=2050$ 元.
- 总的销售所得 $100+50\times80+110\times30=7400$ 元, 与供货商和零售商分账之和不相等.

由于零售商、供货商的日常经营活动并不仅仅是销售单一商品, 仅从账户余额的变动是难以发现问题的, 更难以找出问题的原因; 通过各自的记账系统能够发现问题, 但是难以快速找出问题的原因, 并且面临了从庞杂的区块链数据抽取所需数据、重建合约执行上下文等一系列的挑战; 此外仅凭记账系统的数据则又陷入到尚未利用区块链技术时费时费力人工对账、相互推诿的状态. 采用本文提出的方法则能够快速定位问题所在. 将供货商的账户 ID 记作 account_supply, 将零售商的账户 ID 记作 account_retailer, 则会产生如下读写操作记录.

- 销售一件 1 元秒杀商品:

setState(account_supply, getState(account_supply)+15).

- 销售一件价格为 50 元/件的促销价格商品:

1) setState(account_supply, getState(account_supply)+37.5).

2) setState(account_ retailer, getState(account_retailer)+12.5).

- 销售一件价格为 110 元/件的商品:

1) setState(account_supply, getState(account_supply)+75)

2) setState(account_ retailer, getState(account_ retailer)+35)

在追溯过程中, 形成如后文图 20 的区块链交易依赖关系图, 发现账户余额向上回溯. 回溯过程中因一笔转账交易出现分叉, 经过重放排除了这比转账交易. 继续回溯, 即可发现造成问题的交易. 从读写操作记录可以看出, 每销售一件商品, 在分账时应该对供货商和零售商的账户分别进行读写操作. 然而, 1 元秒杀商品的读写操作记录中仅包含了对供货商账户的操作, 缺少了对零售商账户的操作. 发现问题后, 通过依赖关系关联起来的区块链交易以及对应的读写操作记录可以为修复 bug 提供参考, 也可以作为解决分歧和追责的证据.

4.4 小 结

通过分析性能测试的结果, 可以看出 OnCR 实现与原有实现相比, 对区块链系统的性能产生了有限的影响, 在满负载状态下所耗费的处理时间并未成倍增加, 反映处理能力的 TPS 也并未折半; 在低负载情况下虽然造成出块延迟的密度分布的尾巴略有变长, 但是仍处于可接受的范围. OffCR 的主要计算是在链下进行的, 在性能方面具有优势, 但是无法利用区块链共识机制带来的健壮性. 从案例研究可以看出, 采用本文提出的方法对区块链交易执行进行追溯, 能够用于修复 bug, 解决分歧和举证, 挽回损失并追究开发方的责任.

5 相关工作

根据我们的调研, 目前尚未有与本方法类似的研究. 在保证智能合约正确运行方面的研究, 可分为智能合约的描述与构建、智能合约的证明与验证、智能合约的安全执行 3 方面.

在智能合约的描述与构建方面, 现有研究旨在通过提供更为精确的智能合约描述与构建方法, 从而减少其中的错误. Biryukov 等人[10]提出了面向金融领域的领域特定语言 Findel, 提供了一种可组合、无二义性的智能合约描述方法. He 等人[11]提出了一种被称作 SPESC 的智能合约的描述语言, 面向计算机领域专家和一般用户, 采用了类似自然语言的语法, 有利于智能合约的协作开发. Mavridou 等人[12]基于有限状态机模型提出了 FSolidM 框架, 用户能够利用其图形界面生成智能合约代码; 王璞巍等人[13]给出了面向合同的智能合约的形式化定义以及一种

通用的智能合约实现方法, 使得无论用户是否有计算机领域的知识, 均能够以相同的方式准确地描述想要构建的智能合约中的承诺.

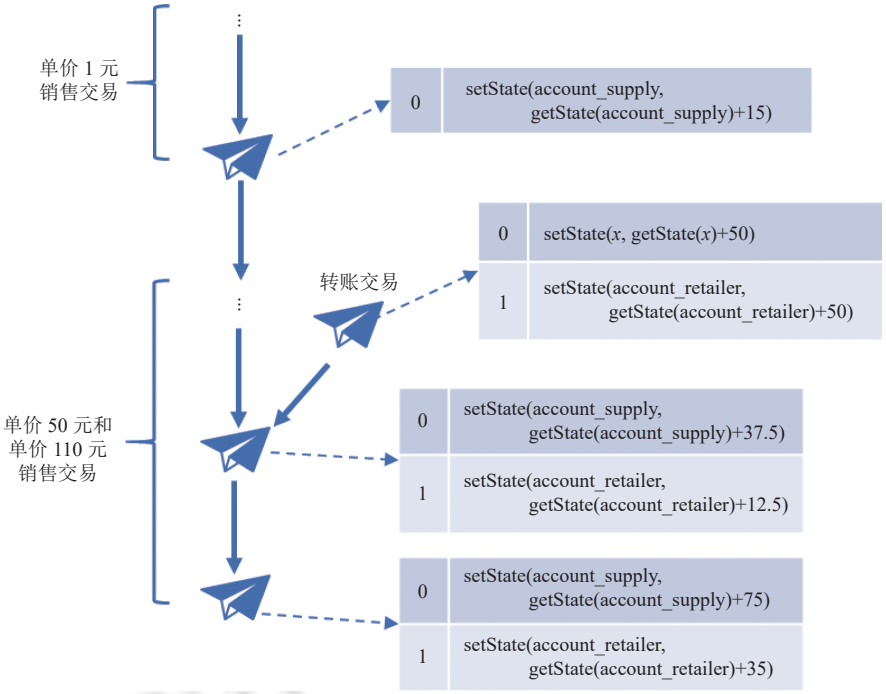


图 20 针对案例的区块链交易执行追溯

智能合约的证明和验证的目的在于确保智能合约代码的安全性和正确性. Dickerson 等人[14]提出将智能合约的形式化证明存储到区块链上, 以便用户对智能合约的正确性进行检查. Kalra 等人[15]提出了 ZEUS 框架, 用于验证合约的正确性和公平性. Hajdu 等人[16]开发了面向 Solidity 智能合约的模块化验证工具, Bartoletti 等人[17]针对 Solidity 合约开发语言提出了一种最小核心演算模型 TinySol, 类似的工作还有 Crafa 等人[18]提出的 Featherweight Solidity, 这类方法旨在识别出智能合约编程语言中最基础的功能, 如转账和外部调用, 从而能够在编译阶段对合约源代码的安全性进行推理. Mavridou 等人[19]提出了 VeriSolid 框架, 开发者能够验证合约并从验证的模型中生成 Solidity 代码. Wang 等人[20]提出的以太坊智能合约缺陷检测方法 ContractWard 利用了机器学习技术. 此外, 在传统软件验证中常用符号执行技术也被引入到智能合约的验证研究中, 研究者主要关注以太坊智能合约的验证[21,22].

在智能合约的执行方面, 主要以保证智能合约正确执行为目标. 朱岩等人[24]在提出的智能合约执行系统中, 利用基于安全多方计算技术来保证智能合约的执行结果是安全可信的, 并能够保护隐私信息的私密性. Lee 等人[25]提出了一种方法来减少 Solidity 智能合约中自动执行的回退函数带来的安全隐患, 提供了新的关键字并修改了 Solidity 编译器以及以太坊虚拟机 EVM 以阻止回退函数的调用. 此外, 部分研究者[26,27]引入软硬件协同的可信执行环境来确保智能合约执行的安全性. 虽然反向调试[28,30,31]和录制重放[32,33]的思路在通用软件调试领域已经得到研究者的重视, 但是尚未被应用到区块链智能合约领域.

上述工作主要从程序开发、运行的正确性方面出发. 然而, 软件安全方面的经验表明, 想要确保智能合约的正确性要比发现其中的错误要困难得多, 特别是在实际应用中, 对智能合约开发的功能需求背后的逻辑普遍较为复杂. 此外, 这些方法也难以保证排除带有恶意开发的能够正确执行的智能合约. 虽然能够建立智能合约文本与合约代码之间的强制性关联关系[38], 然而这是一种静态的关系, 在真实应用场景中难以建立合约代码与链上实际执行结果之间的联系. 本文提出的方法能够提升智能合约部署后在生产环境中运行过程中的可调式性, 有利于解决智能合约缺陷带来的问题.

6 总结与展望

针对区块链交易执行追溯问题, 提出了一种基于录制重放的区块链交易执行追溯方法. 通过增强合约容器建立录制重放机制, 能够录制对区块链世界状态的读写操作供之后的离线分析和重放执行. 提出了基于世界状态读写的交易依赖分析方法, 能够基于交易读写键集合的重叠情况识别交易之间的依赖形成有向无环图, 通过在有向无环图上按需逐条回溯进行重访和分析实现了对区块链交易执行过程的追溯. 提出了读写操作记录可验证机制, 无论是将读写操作记录存储于链上还是链下, 均能验证给定的读写操作与真实执行过程的一致性. 实验中对比了区块链合约容器原有实现和引入录制重放方法后性能差异, 发现即使采用性能影响最大的链上存储方案, 额外性能消耗处于可接受范围内. 实验中还比较了链上存储和链下存储方案性能差异. 通过案例研究展示了所提出方法的有效性和优点.

录制的读写操作记录采用链上存储会对区块链系统的性能产生影响, 采用链下存储则无法避免数据丢失的问题, 针对这一问题, 在下一步工作中拟研究基于分布式冗余存储的链下存储方案, 在提高性能的同时保障合约执行日志被妥善保存.

References:

[1] Szabo N. Smart contracts. 1994. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html

[2] Xia Q, Dou WS, Guo KW, Liang G, Zuo C, Zhang FJ. Survey on blockchain consensus protocol. Ruan Jian Xue Bao/Journal of Software, 2021, 32(2): 277–299 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6150.htm [doi: 10.13328/j.cnki.jos.006150]

[3] Ouyang LW, Wang S, Yuan Y, Ni XC, Wang FY. Smart contracts: Architecture and research progresses. Acta Automatica Sinica, 2019, 45(3): 445–457 (in Chinese with English abstract). [doi: 10.16383/j.aas.c180586]

[4] Guan ZT, Lu X, Yang WT, Wu LF, Wang NY, Zhang ZJ. Achieving efficient and privacy-preserving energy trading based on blockchain and ABE in smart grid. Journal of Parallel and Distributed Computing, 2021, 147: 34–45. [doi: 10.1016/j.jpdc.2020.08.012]

[5] Ur Rahman M, Guidi B, Baiardi F. Blockchain-based access control management for decentralized online social networks. Journal of Parallel and Distributed Computing, 2020, 144: 41–54. [doi: 10.1016/j.jpdc.2020.05.011]

[6] Tan HB, Zhou T, Zhao H, Zhao Z, Wang WD, Zhang ZX, Sheng NZ, Li XF. Archival data protection and sharing method based on blockchain. Ruan Jian Xue Bao/Journal of Software, 2019, 30(9): 2620–2635 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5770.htm [doi: 10.13328/j.cnki.jos.005770]

[7] Jin H, Xu C, Luo Y, Li PL, Cao Y, Mathew J. Toward secure, privacy-preserving, and interoperable medical data sharing via blockchain. In: Proc. of the 25th Int'l Conf. on Parallel and Distributed Systems. Tianjin: IEEE, 2019. 852–861. [doi: 10.1109/ICPADS47876.2019.00126]

[8] Xu XW, Pautasso C, Zhu LM, Lu QH, Weber I. A pattern collection for blockchain-based applications. In: Proc. of the 23rd European Conf. on Pattern Languages of Programs. Irsee: ACM, 2018. 3. [doi: 10.1145/3282308.3282312]

[9] Torres CF, Steichen M, State R. The art of the scam: Demystifying honeypots in Ethereum smart contracts. In: Proc. of the 28th USENIX Conf. on Security Symp. Santa Clara: USENIX Association, 2019. 1591–1607.

[10] Biryukov A, Khovratovich D, Tikhomirov S. Findel: Secure derivative contracts for Ethereum. In: Proc. of the 2017 Int'l Conf. on Financial Cryptography and Data Security. Sliema: Springer, 2017. 453–467. [doi: 10.1007/978-3-319-70278-0_28]

[11] He X, Qin BH, Zhu Y, Chen X, Liu Y. SPESC: A specification language for smart contracts. In: Proc. of the 42nd Annual Computer Software and Applications Conf. Tokyo: IEEE, 2018. 132–137. [doi: 10.1109/COMPSAC.2018.00025]

[12] Mavridou A, Laszka A. Designing secure Ethereum smart contracts: A finite state machine based approach. In: Proc. of the 22nd Int'l Conf. on Financial Cryptography and Data Security. Nieuwpoort: Springer, 2018. 523–540. [doi: 10.1007/978-3-662-58387-6_28]

[13] Wang PW, Yang HT, Meng J, Chen JC, Du XY. Formal definition for classical smart contracts and reference implementation. Ruan Jian Xue Bao/Journal of Software, 2019, 30(9): 2608–2619 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5773.htm [doi: 10.13328/j.cnki.jos.005773]

[14] Dickerson T, Gazzillo P, Herlihy M, Saraph V, Koskinen E. Proof-carrying smart contracts. In: Proc. of the 2018 Int'l Conf. on Financial Cryptography and Data Security. Nieuwpoort: Springer, 2018. 325–338. [doi: 10.1007/978-3-662-58820-8_22]

[15] Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing safety of smart contracts. In: Proc. of the 2018 Network and Distributed Systems Security (NDSS) Symp. San Diego, 2018. [doi: 10.14722/ndss.2018.23082]

[16] Hajdu Á , Jovanović D. SOLC-VERIFY: A modular verifier for Solidity smart contracts. In: Proc. of the 11th Int'l Conf. New York: Springer, 2019. 161–179. [doi: 10.1007/978-3-030-41600-3_11]

[17] Bartoletti M, Galletta L, Murgia M. A minimal core calculus for Solidity contracts. In: Proc. of the 2019 ESORICS Int'l Workshops. Luxembourg: Springer, 2019. 233–243. [doi: 10.1007/978-3-030-31500-9_15]

[18] Crafa S, Di Pirro M, Zucca E. Is Solidity solid enough? In: Proc. of the 2019 Int'l Conf. on Financial Cryptography and Data Security. St. Kitts: Springer, 2019. 138–153. [doi: 10.1007/978-3-030-43725-1_11]

[19] Mavridou A, Laszka A, Stachtiari E, Dubey A. VeriSolid: Correct-by-design smart contracts for Ethereum. In: Proc. of the 23rd Int'l Conf. on Financial Cryptography and Data Security. St. Kitts: Springer, 2019. 446–465. [doi: 10.1007/978-3-030-32101-7_27]

[20] Wang W, Song JJ, Xu GQ, Li YD, Wang H, Su CH. ContractWard: Automated vulnerability detection models for Ethereum smart contracts. IEEE Trans. on Network Science and Engineering, 2021, 8(2): 1133–1144. [doi: 10.1109/TNSE.2020.2968505]

[21] Permenev A, Dimitrov D, Tsankov P, Drachsler-Cohen D, Vechev M. VerX: Safety verification of smart contracts. In: Proc. of the 2020 IEEE Symp. on Security and Privacy. San Francisco: IEEE, 2020. 1661–1677. [doi: 10.1109/SP40000.2020.00024]

[22] Chang JL, Gao B, Xiao H, Sun J, Cai Y, Yang ZJ. sCompile: Critical path identification and analysis for smart contracts. In: Proc. of the 21st Int'l Conf. on Formal Engineering Methods. Shenzhen: Springer, 2019. 286–304. [doi: 10.1007/978-3-030-32409-4_18]

[23] EPFL-LARA. Stainless for smart contracts. 2020. https://github.com/epfl-lara/smart

[24] Zhu Y, Song XX, Xue XB, Qin BH, Liu GW. Smart contract execution system over blockchain based on secure multi-party computation. Journal of Cryptologic Research, 2019, 6(2): 246–257 (in Chinese with English abstract). [doi: 10.13868/j.cnki.jcr.000299]

[25] Lee S, Cho ES. A modified smart contract execution enviroment for safe function calls. In: Proc. of the 43rd IEEE Annual Computer Software and Applications Conf. (COMPSAC). Milwaukee: IEEE, 2019. 904–907. [doi: 10.1109/COMPSAC.2019.00135]

[26] Cheng R, Zhang F, Kos J, He W, Hynes N, Johnson N, Juels A, Miller A, Song D. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In: Proc. of the 2019 IEEE European Symp. on Security and Privacy. Stockholm: IEEE, 2019. 185–200. [doi: 10.1109/EuroSP.2019.00023]

[27] Müller C. Execution of smart contracts with ARM TrustZone [Ph.D. Thesis]. Bern: University of Bern, 2019.

[28] Spinellis D. Modern debugging: The art of finding a needle in a haystack. Communications of the ACM, 2018, 61(11): 124–134. [doi: 10.1145/3186278]

[29] Sousa TB, Ferreira HS, Correia FF, Aguiar A. Engineering software for the cloud: Messaging systems and logging. In: Proc. of the 22nd European Conf. on Pattern Languages of Programs. Irsee: ACM, 2017. 14. [doi: 10.1145/3147704.3147720]

[30] Cui WD, Ge XY, Kasikci B, Niu B, Sharma U, Wang RY, Yun I. REPT: Reverse debugging of failures in deployed software. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 17–32.

[31] Hoshino S, Arahori Y, Gondow K. STRAB: State recovery using reverse execution at IR level for concurrent programs. In: Proc. of the 36th Annual ACM Symp. on Applied Computing. Online: ACM, 2021. 1532–1541. [doi: 10.1145/3412841.3442028]

[32] Pobee E, Chan WK. AggrePlay: Efficient record and replay of multi-threaded programs. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 567–577. [doi: 10.1145/3338906.3338959]

[33] Zuo GF, Ma JC, Quinn A, Bhatotia P, Fonseca P, Kasikci B. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In: Proc. of the 42nd ACM SIGPLAN Int'l Conf. on Programming Language Design and Implementation. ACM, 2021. 1155–1170. [doi: 10.1145/3453483.3454101]

[34] Jiang YY, Xu C, Ma XX, Lü J. Approaches to obtaining shared memory dependences for dynamic analysis of concurrent programs: A survey. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 747–763 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5193.htm [doi: 10.13328/j.cnki.jos.005193]

[35] Li CX, Chen S, Zheng LS, Zuo C, Jiang BY, Liang G. RepChain—A permissioned blockchain toolkit implemented by reactive programming. Ruan Jian Xue Bao/Journal of Software, 2019, 30(6): 1670–1680 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5743.htm [doi: 10.13328/j.cnki.jos.005743]

[36] Bartoletti M, Pompianu L. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In: Proc. of the 2017 Int'l Conf. on Financial Cryptography and Data Security. Sliema: Springer, 2017. 494–509. [doi: 10.1007/978-3-319-70278-0_31]

[37] Liu P, Zhang XY, Tripp O, Zheng YH. Light: Replay via tightly bounded recording. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 55–64. [doi: 10.1145/2737924.2738001]

[38] Mohanty D. R3 Corda for Architects and Developers. Berkeley: Apress, 2019. 49–60. [doi: 10.1007/978-1-4842-4529-3]

附中文参考文献:

[2] 夏清, 窦文生, 郭凯文, 梁赓, 左春, 张凤军. 区块链共识协议综述. 软件学报, 2021, 32(2): 277–299. http://www.jos.org.cn/1000-9825/6150.htm [doi: 10.13328/j.cnki.jos.006150]

[3] 欧阳丽炜, 王帅, 袁勇, 倪晓春, 王飞跃. 智能合约: 架构及进展. 自动化学报, 2019, 45(3): 445–457. [doi: 10.16383/j.aas.c180586]

[6] 谭海波, 周桐, 赵赫, 赵哲, 王卫东, 张中贤, 盛念祖, 李晓风. 基于区块链的档案数据保护与共享方法. 软件学报, 2019, 30(9): 2620–2635. http://www.jos.org.cn/1000-9825/5770.htm [doi: 10.13328/j.cnki.jos.005770]

[13] 王璞巍, 杨航天, 孟佶, 陈晋川, 杜小勇. 面向合同的智能合约的形式化定义及参考实现. 软件学报, 2019, 30(9): 2608–2619. http://www.jos.org.cn/1000-9825/5773.htm [doi: 10.13328/j.cnki.jos.005773]

[24] 朱岩, 宋晓旭, 薛显斌, 秦博涵, 刘国伟. 基于安全多方计算的区块链智能合约执行系统. 密码学报, 2019, 6(2): 246–257. [doi: 10.13868/j.cnki.jcr.000299]

[34] 蒋炎岩, 许畅, 马晓星, 吕建. 获取访存依赖: 并发程序动态分析基础技术综述. 软件学报, 2017, 28(4): 747–763. http://www.jos.org.cn/1000-9825/5193.htm [doi: 10.13328/j.cnki.jos.005193]

[35] 李春晓, 陈胜, 郑龙帅, 左春, 蒋步云, 梁赓. 响应式许可链基础组件——RepChain. 软件学报, 2019, 30(6): 1670–1680. http://www.jos.org.cn/1000-9825/5743.htm [doi: 10.13328/j.cnki.jos.005743]

陈胜(1972－), 男, 高级工程师, 主要研究领域为区块链核心组件架构, 区块链技术的应用.

左春(1959－), 男, 研究员, 主要研究领域为软件工程.

方明哲(1989－), 男, 博士, 工程师, 主要研究领域为区块链.

李玉成(1961－), 男, 研究员, 主要研究领域为并行软件, 计算科学.

蒋步云(1972－), 男, 高级工程师, 主要研究领域为区块链.

梁赓(1962－), 男, 正高级工程师, 主要研究领域为区块链, 计算机应用.

李春晓(1968－), 女, 高级工程师, CCF 专业会员, 主要研究领域为区块链, 数字博物馆, 数字图书馆.