计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220887
60(2): 311−325, 2023

智能合约 Gas 优化综述

宋书玮　倪孝泽　陈　厅
（电子科技大学计算机科学与工程学院　成都　611731）
（shuwei@std.uestc.edu.cn）

Gas Optimization for Smart Contracts: A Survey

Song Shuwei, Ni Xiaoze, and Chen Ting
（*School of Computer Science and Engineering*, *University of Electronic Science and Technology of China*, *Chengdu* 611731）

Abstract　The most significant feature of Blockchain 2.0 is the introduction of support for smart contracts, which enables the blockchain to run various applications. The smart contract is a type of computer software that runs automatically according to pre-defined code logic. Distinguished from traditional software, smart contracts are empowered by blockchain technology with the ability to execute correctly on mutually untrusted nodes without relying on a trusted central authority, making them widely used in areas such as digital payments and the sharing economy. To prevent the waste of computing resources caused by the abuse of smart contracts, blockchains such as Ethereum charges Gas fees for two activities, deployment and execution of smart contracts. The computing resource consumed by smart contracts is the factor that determines the cost. Smart contracts with inefficient code are wasteful of resources and vulnerable to attacks, and the developers and users of them suffer unnecessary costs. Therefore, optimizing smart contracts to save resources has become a critical issue for developers and researchers. This survey first analyzes the main challenges of Gas optimization for smart contracts in detail, and then reviews and summarizes the various optimization techniques proposed in recent years. Finally, we discuss future work, which provides references for developers and researchers who explore smart contracts.

Key words　blockchain; smart contract; Gas optimization; code efficiency; software analysis

摘　要　区块链 2.0 最显著的特征是增加了对智能合约的支持，这使得区块链拥有了运行各种应用程序的能力. 智能合约是一种根据预先定义的代码逻辑自动运行的计算机软件. 区别于传统软件，区块链技术赋予了智能合约不依赖可信中心机构而在相互不信任的节点上正确执行的能力，使其在数字支付、共享经济等领域被广泛地应用. 为了防止滥用智能合约导致计算资源被浪费，以太坊等区块链向部署和执行智能合约这 2 种活动收取 Gas（燃料）费用. 智能合约消耗的计算资源是决定费用高低的因素. 具有低效代码的智能合约浪费资源且易受攻击，此类智能合约的开发者和用户将承担不必要的费用. 因此，优化智能合约以节省资源已经成为开发者和研究者重点关注的问题. 首先详细分析了智能合约 Gas 优化所面临的主要挑战；然后回顾和总结了近年来提出的各种优化技术；最后展望了该研究方向的未来工作，旨在为智能合约的开发者和研究人员提供参考和借鉴.

关键词　区块链；智能合约；Gas 优化；代码效率；软件分析

中图法分类号　TP391

收稿日期：2022−10−21；**修回日期**：2022−12−21

基金项目：国家自然科学基金项目（61872057, U19A2066）；四川省自然科学基金项目（2022NSFSC0871）
This work was supported by the National Natural Science Foundation of China (61872057, U19A2066) and the Natural Science Foundation of Sichuan Province (2022NSFSC0871).

通信作者：陈厅（brokendragon@uestc.edu.cn）

区块链的概念最早在2008年作为比特币的底层技术被提出[1]. 最初, 区块链被视为一个分布式的公共账本, 记录虚拟货币的所有交易. 这个阶段的区块链仅局限于虚拟货币的开发和买卖. 然而, 随着近年来区块链技术的快速发展, 它所具备的去中心化、防篡改、匿名、可审计等特性引起了学术界和工业界的广泛关注, 区块链技术被认为可以应用于除虚拟货币以外更多的领域[2], 其中一个很有前景的应用是智能合约(smart contract).

对智能合约的支持标志着区块链进入2.0时代[2]. 智能合约是一种根据预先定义的代码逻辑在区块链的多个节点上同步运行的计算机软件[3]. 区块链赋予智能合约不依赖可信中心机构而在相互不信任的节点上正确执行的能力[3]. 由于智能合约具备去中心化、抗伪造等传统计算机软件不具备的独特优势, 它有潜力重塑银行、保险和内容平台等行业[4]. 目前, 在最流行的支持智能合约的区块链以太坊[5]中, 已经部署了超过4 400万个智能合约[6]. 因此, 本文主要关注以太坊以及其上部署的智能合约, 下文中若无特殊说明, 则区块链特指以太坊区块链, 智能合约指部署在以太坊上的智能合约.

开发者通常使用高级语言(例如Solidity和Vyper)编写智能合约, 开发完成后, 将其编译成能够在以太坊虚拟机(Ethereum virtual machine, EVM)中运行的EVM字节码. 然后, 通过发送一笔特殊的交易将智能合约的字节码部署到区块链上. 用户可以发送交易来调用已经部署的智能合约, 交易中应当指定用户想调用的智能合约、接口, 并携带接口所需的参数. 根据以太坊黄皮书的规定, 区块链网络中每个节点都应维护一份相同的区块链副本, 且都应执行存储在区块链中的所有交易[7]. 这表明部署后的智能合约被存储在所有节点的磁盘中, 被调用的智能合约在所有节点的EVM中执行. 综上所述, 部署和调用智能合约会消耗区块链网络中每个节点的计算资源(CPU、磁盘空间、内存等). 因此, 攻击者可以通过部署大量智能合约或者发起大量交易调用智能合约对区块链实施拒绝服务攻击[8].

为了应对上述的资源滥用和攻击, 以EVM作为智能合约执行环境的区块链(例如以太坊、BNB Smart Chain[9]和polygon[10]等)采用了燃料(Gas)机制向部署和调用智能合约这两种活动收取费用. 具体来说, 区块链根据被部署智能合约的大小和被调用的智能合约中执行的指令, 分别向开发者和用户收取费用. 体积大的智能合约占用更多存储空间, 复杂的指令相较简单的指令消耗更多资源, 因此需要支付更多费用.

Gas机制提高了攻击者实施拒绝服务攻击的成本, 有效地防止了区块链资源被滥用. 然而, 它同时也给智能合约的开发引入了新的挑战. 由于开发者缺乏经验、不了解Gas机制、编译器优化不足和缺少辅助工具等原因, 许多代码效率低下的智能合约被部署到区块链[11]. 此类智能合约实际消耗的费用大于其真正必需的费用, 因此造成资源浪费. 一方面, 部署智能合约时, 未经优化的冗余代码消耗更多存储空间, 开发者将支付更多费用; 另一方面, 部署后的智能合约中低效率的代码可能被调用无数次, 这将浪费大量资源, 用户将支付更多费用; 除此之外, 未经优化的智能合约容易遭受攻击(详见2.2节). 因此, 研究智能合约优化方法具有重要的现实意义.

目前已经出现了一些智能合约优化方法, 它们能够从源代码、字节码等不同角度优化智能合约从而降低Gas费用. 基于目前研究进展, 本文对现有的智能合约优化方法进行回顾与总结, 概括出3个研究挑战和3类典型的优化方法, 希望能够给当前和未来的智能合约优化相关研究提供一定的参考.

1 相关背景知识

1.1 区块链、以太坊和智能合约

2008年, Nakamoto[1]在其论文中提出了比特币和一种点对点电子交易系统, 该系统中的去中心化共享账本通过密码学方法保证其不可篡改和不可伪造. 2009年1月, 该系统首次发布, 标志着比特币的诞生. 此后, 比特币因具备优于传统货币的一些优良特性(例如去中心化、匿名、免监管和全世界流通等)而迅速吸引了大量关注.

区块链是比特币、以太币以及类似数字加密货币的基础, 它本质上是若干数据区块链接而成的一种链状数据结构, 并且使用数字签名、加密哈希和分布式共识等算法来实现去中心化、可审计等特性[12]. 其中每个数据区块记录时间戳、若干交易和前一个区块的哈希值[1]. 区块链不存储在某个中心化的服务器, 而是存储于P2P网络中的每个节点, 即每个节点都存储一份区块链的副本并通过同步使副本之间保持一致[3]. 为了与其他节点达成共识, 每个节点都会执行提交到区块链上的交易[7]. 节点之间没有上下级关系(即没有中心节点)并且它们的地理位置是分散的, 任何节点都可以加入或者退出该P2P网络. 这表

明所有的数据都是公开的，并且容易被任何节点验证. 因此，除非取得所有节点的一致同意，否则很难修改区块链中记录的交易[3]. 区块链技术使得互不信任的双方可以在不借助可信第三方的情况下安全地进行交易.

尽管数字加密货币是区块链最早、最著名的应用，但区块链的应用不止于此. 智能合约的出现使得基于区块链的更复杂的应用得以实现. 智能合约一词最早在 1997 年被 Szabo[13] 提出，他认为将法律合同中的条款编写成代码，使其自动、强制地执行，可以减少成本并避免恶意行为. 在其论文中，智能合约指使用软件来实现和执行的法律合同. 时至今日，智能合约在不同学科中有着不同的含义，其中一个概念被广泛采用，本文也基于此概念展开研究：智能合约是根据预先定义的代码逻辑在区块链网络的多个节点上自动、同步运行的软件[14]. 智能合约具有 4 点特性[13]：1）自动，即智能合约由交易触发，然后自动地执行；2）自治，即一旦被触发，就无法阻止智能合约执行；3）透明，即智能合约对区块链网络中的每个节点都是透明的；4）灵活，即智能合约可以根据不同的场景需求进行调整.

以太坊（Ethereum）自 2015 年发布以来已经逐步发展为最流行的支持智能合约的区块链平台[3]. 以太坊采用与传统银行系统类似的账户模型[15]，它包含 2 个类型的账户：由公私钥对控制的外部所有账户（external owned account, EOA）和由智能合约代码控制的合约账户（contract account）. 两类账户的主要区别在于：1）EOA 可以发送交易以调用智能合约，合约账户无法主动与 EOA 交互，但它可以在执行过程中调用其他智能合约；2）合约账户包含智能合约代码而 EOA 没有.

智能合约的生命周期通常包含开发、编译、部署、调用、执行和销毁 6 个阶段.

1）开发阶段. 开发者使用编程语言（例如：Solidity 和 Vyper）开发智能合约. 由于开发者的水平、经验和目的不同，导致开发出的智能合约可能存在代码效率低、包含恶意行为等问题. 2）编译阶段. 智能合约的源代码被编译成字节码. 字节码由超过 150 种指令排列组合而成[7]，根据指令操作的对象可以将指令分为如表 1 所示的 5 类[8]. 每种指令的功能不一样，因而执行它们所消耗的计算资源也不同. 3）部署阶段. 开发者发起一笔接收者为空的交易，该交易将被区块链视为合约部署请求. 区块链将创建一个新的合约账户来存储交易中携带的智能合约代码. 4）调用阶段. 当交易的接收者是一个合约账户时，该交易被视为智能合约调用请求. 根据交易中指定的函数和参数，事先部署的智能合约代码将被执行. 5）执行阶段. 以太坊提供一个基于栈的图灵完备的虚拟机（EVM）来执行智能合约. EVM 中包含 3 种存储结构以辅助智能合约的执行：用于存放局部变量的栈（stack）、用于存放函数参数和返回值的内存（memory）和用于持久化存放数据的存储（storage）. 其中，栈和内存是非持久化的，它们在完成一次交易后将被清除，因此存储的空间相较于栈和内存更加昂贵. 6）销毁阶段. 如果开发者计划在未来销毁智能合约，可以在开发阶段添加“自毁”代码，以便于从区块链中抹除该合约账户.

Table 1　Five Types of Instructions for EVM

表 1　以太坊虚拟机的 5 类指令

类别	定义
第 1 类	不操作任何数据结构的指令，例如 JUMPDEST.
第 2 类	进行栈操作的指令，例如 PUSHx，或者对栈中已有的值进行操作的指令，例如 ADD.
第 3 类	从区块链或交易中获取信息的指令，例如 TIMESTAMP 和 ORIGIN.
第 4 类	用于读写内存的指令，例如 MSTORE.
第 5 类	用于读写存储的指令，例如 SSTORE.

1.2　Gas 机制

如 1.1 节所述，区块链网络中每个节点都应存储区块链的完整副本，并执行历史上所有的交易以进行同步，如此重复的存储和执行导致部署和调用智能合约会消耗区块链网络中每个节点的计算资源（例如 CPU、磁盘空间、内存等）. 为了防止计算资源被故意或无意地滥用，以太坊引入了 Gas 机制，即根据交易消耗的计算资源向交易发送方收取费用，执行交易所需的计算资源越多，那么交易发送方需要支付的费用就越多[8]. 交易发送方在发起交易时应该指定愿意为此交易支付的 Gas 总限额（gas limit），如果该限额比执行智能合约时实际花费的 Gas 小，则会产生 Gas 耗尽异常（out-of-gas exception）[8]，使得交易失败. 一旦一条交易失败，区块链将恢复到执行该交易之前的状态，但是交易发起者支付的 Gas 不会被退还，因为执行该交易已经消耗了节点的计算资源[16]. Liu 等人[17] 的研究发现 Gas 耗尽异常在以太坊所有异常情况中占据 90% 以上，这在一定程度上表明：Gas 机制阻止了大量的意图以不合理的/不够的费用执行交易的用户和攻击者.

Gas 机制主要从 3 个方面发挥作用[18]：1）向交易

发起者收取费用可以抑制攻击者通过发起毫无价值的交易来浪费节点的计算能力;2)通过设置较高的Gas消耗,抑制用户使用过多的存储空间;3)使交易一定会终止,从而防止基于非终止交易的拒绝服务攻击.

交易费用是Gas价格(gas price)和Gas消耗(gas cost)的乘积,其中Gas价格是受市场影响的动态值,它指交易发起者愿意为每个单位的Gas所支付的价格.例如,2022年9月23日的平均Gas价格为1.306×10^{-8}以太币[19],约等于1.7×10^{-5}美元[20].Gas价格在不断波动当中,峰值价格有时甚至能够达到往常的百倍以上[19].影响交易费用的另一因素是Gas消耗,是由以太坊规定的静态值,每个EVM指令都有其对应的Gas消耗值,它是指执行某个指令需要多少单位的Gas[8].以太坊将指令的Gas消耗与执行该指令所需的计算资源设计成正比.如表2所示,有些指令所需的Gas消耗相对较少(例如ADD, POP和AND等),因为它们只涉及简单的操作,而另一些指令Gas消耗较高(例如用于创建账户的指令CREATE以及用于写入存储的指令SSTORE,值得注意的是,SSTORE指令将一个非零的值写入存储中原本为零值的位置时,它消耗20 000个单位的Gas;其余情况消耗5 000个单位的Gas).此外,部署智能合约的开发者也需要支付Gas,该费用与智能合约字节码长度成正比[8].

综上所述,交易发起者需要为部署和调用智能合约付费,例如向栈顶压入一个数需要3个单位的Gas(约等于5.1×10^{-5}美元).由于智能合约中包含若干条指令并且每个智能合约可能被调用无数次,所以即便一条指令所需的费用较低,智能合约所消耗的总费用也可能增长为一个庞大的数目.为了展示交易费用的规模及其影响,Albert等人[16]统计了2017—2019年用于交易的资金,调查结果显示该数值达到了1.57亿美元.

Table 2 Gas Cost for Some Instructions [7]

表2 某些指令的Gas消耗[7]

指令	描述	Gas消耗量
ADD, SUB	算数运算	3
MUL, DIV		5
AND, OR, XOR	逻辑运算	3
LT, GT, SLT, SGT, EQ	比较运算	3
POP	栈操作	2
PUSH, DUP, SWAP		3
JUMP	无条件跳转	8
JUMPI	有条件跳转	10
MLOAD, MSTORE	内存操作	3
SLOAD	存储操作	200
SSTORE		5 000或20 000
CREATE	创建账户	32 000

2 Gas优化的动机与挑战

具备完全相同功能的智能合约可能有多种不同的实现,因而部署和调用它们所需要的Gas也不尽相同.由于Gas与金钱直接相关,用户自然地愿意选择使用功能完整且成本更低的智能合约,进而促使开发者对智能合约进行优化以减少Gas开销,然而一些挑战阻碍了开发者完成这一目标.

2.1 低效的智能合约

受到包括但不限于开发者的开发经验、编程习惯、使用的编程语言、开发工具等多种因素的影响,具备完全相同功能的智能合约可能有多种不同的实现.例如,Liao等人[21]对智能合约的大规模实证研究发现智能合约中某些由高级语言编写的代码片段可以用内联汇编替代,但不会改变或影响原智能合约的功能.

图1中展示了2个具备完全相同功能的智能合约,即图1(a)中的ExpensiveSampleContract和图1(b)中的CheapSampleContract.首先,它们的第2行代码都定义了一个无符号整数类型的全局变量*count*.值得注意的是,智能合约的全局变量被存放在存储中.其次,从2个智能合约的第3行代码开始,它们都定义了一个函数用于循环执行某种操作(例如:每次循环中若满足某种条件就向一个账户转移以太币),并且*count*被用于记录该操作执行的次数.2个智能合约的差别在于:ExpensiveSampleContract直接在循环中更新*count*;而CheapSampleContract在循环操作之前先将*count*的值赋给一个临时变量*temp*,然后在循环结束后将更新后*temp*的值赋给*count*.因此,CheapSampleContract的代码包含更多指令,并且运行时需要额外的空间来存储局部变量*temp*.相较之下,ExpensiveSampleContract似乎是相同功能下更优的一种实现.

然而,请注意1.1节中提及的内容:存储空间用于持久化存放数据并且每个节点都应维护副本,而栈和内存中的数据是非持久化的,因此存储资源更昂贵.如表2所示,存储操作的Gas消耗远远超过栈操作和内存操作.从消耗资源的角度分析,Expensive-

```
1 contract ExpensiveSampleContract {
2    uint count;
3    function expensiveFunc(uint n) {
4       for (uint i = 0; i < n; i++) {
5          // other operations
6          count++;
7       }
8    }
9 }
```

(a) 相对浪费Gas的智能合约

```
1 contract CheapSampleContract {
2    uint count;
3    function cheapFunc(uint n) {
4       uint temp = count;
5          for (uint i = 0; i < n; i++) {
6          // other operations
7          temp++;
8       }
9       count = temp;
10   }
11 }
```

(b) 相对节省Gas的智能合约

Fig. 1 Two smart contracts with identical functionality

图 1 功能完全相同的 2 个智能合约

SampleContract 分别读取和写入存储 n 次，而 CheapSampleContract 将该读取或写入的次数降低至 1. 此外，由于新增的局部变量 *temp* 存放在栈中，所以 CheapSampleContract 对栈的读取和写入分别增加 n+1 次(循环内 n 次，循环外 1 次). 假设读取并写入存储一次需消耗 C_{Storage} 个单位的 Gas，读取并写入栈一次需消耗 C_{Stack} 个单位的 Gas，则 CheapSampleContract 相比 ExpensiveSampleContract 能够节省的 Gas 数量为

$$(n-1)C_{\text{Storage}}-(n+1)C_{\text{Stack}}. \tag{1}$$

因为 $C_{\text{Storage}} \gg C_{\text{Stack}}$，因此式(1)的值大于零，即 CheapSampleContract 相对消耗更少的 Gas. 为证明上述推断，本文使用最新的编译器 Solidity 0.8.17 编译 2 个智能合约，并将它们部署到搭建的私有以太坊区块链上. 然后，在交易中分别将参数 n 设置为 100, 500, 1 000，并调用 2 个智能合约以测试 Gas 消耗量. 重复上述测试过程 10 次，并对测试结果求平均值以减少偶然产生的误差. 如表 3 所示，CheapSampleContract 始终消耗更少的 Gas.

综上所述，CheapSampleContract 大幅减少了 Gas 消耗极高的存储操作，而且只引入了开销很小的栈操作，从而可以为调用该合约的用户节省交易费用. 虽然字节码长度的增加会导致部署成本升高，但由于部署是一次性的活动，增加少量成本以吸引用户对开发者来说是可以接受的.

本文称类似于 ExpensiveSampleContract 的智能合约为“低效的”. 具体来说，如果一个智能合约存在另一种功能完全相同并且更节省 Gas 的实现，则本文将其视为低效的智能合约，这与另一些研究中定义的“优化不足的智能合约”[22]“Gas 效率低的智能合约”[11] 具有相同的含义. Chen 等人[22] 提出：智能合约低效的原因是其中包含 Gas 消耗大(gas-costly)的指令序列，该指令序列可以被另一些消耗更少 Gas(包括一次部署和多次执行所消耗的 Gas)的指令序列替代而不影响原本的语义[23].

Table 3 Comparison of Gas Consumption of Two Smart Contracts

表 3 2 个智能合约的 Gas 消耗量对比

循环次数	Gas 消耗量	
	CheapSampleContract	ExpensiveSampleContract
100	75 896	80 320
500	204 303	295 954
1 000	347 681	582 554

2.2 Gas 优化的动机

2.1 节中提到低效的智能合约存在另一种更节省 Gas 的实现，这意味着低效的智能合约实际消耗的 Gas 数量超过完成其功能必需的数量. 随之带来的问题包括：1)开发者和用户的金钱被浪费，因为 Gas 与金钱直接相关；2)区块链的计算资源被浪费，因为 Gas 消耗反映计算资源的消耗. 受高交易费用影响的用户可能不愿意长期使用智能合约，进而使智能合约开发商的收入不能得到保证. 此外，对计算资源的浪费可能引发环保问题. 综上，低效的智能合约将阻碍区块链的健康持续发展.

如果低效的智能合约是少数，那么上述 2 个问题尚不足以带来严重影响. 不幸的是 1)相关研究表明大部分的智能合约都是低效的[11]；2)据统计，以太坊一天之内(2019 年 5 月 16 日)产生的 Gas 费用就超过了 2.5 亿美元[24]. 这些调查结果极大地促使了从业者和研究者关注 Gas 优化，即用高效的智能合约替换原本低效的版本以节省 Gas、节省计算资源以及节省使用区块链的成本.

对智能合约进行优化能够提高攻击者发起拒绝服务攻击[25] 的难度. 具体而言，攻击者发起交易、设置了智能合约的状态后，将使得后续对该合约发起的交易以一定概率产生 Gas 耗尽异常. 例如，攻击者往一个数组中添加大量元素，将使得后续对该数组进行遍历需要更多 Gas，如果用户调用该合约时仍然按常规设置 Gas 总限额，则可能产生 Gas 耗尽异常. 为缓解上述拒绝服务攻击，Gas 优化通过降低执行智

能合约的 Gas 消耗量, 提升产生 Gas 耗尽异常的阈值, 从而增加攻击难度.

2.3 Gas 优化的挑战

研究者认为优化智能合约的代码在节省成本方面具有巨大的潜力[24], 然而实施 Gas 优化并非易事, 它面临诸多方面的挑战, 本节总结为 3 个方面:

1)智能合约与传统计算机程序的优化具有显著差异, 所以不能将已有的优化方法直接应用到智能合约上. 以图 1 中的 2 个智能合约为例, 传统的优化方法会认为 ExpensiveSampleContract 相对 CheapSampleContract 更优, 所以无需优化, 主要原因为: ①传统程序中, 访问局部变量和全局变量的开销没有明显区别, 而在智能合约中访问全局变量的成本极高; ②ExpensiveSampleContract 少使用了一个局部变量(即 *temp*), 因此运行时占用的内存更少; ③由于 ExpensiveSampleContract 包含相对更少的步骤(即无需对局部变量 *temp* 赋值和把 *temp* 的值赋给全局变量 *count*), 所以它的字节码中包含相对更少的指令, 于是存放其代码所需的磁盘空间更少.

2)开发高效的智能合约对开发者提出了较高的要求. 为了开发高效的智能合约, 开发者通常需要采用有利于节省 Gas 的编程模式, 深入了解 EVM 字节码、不同指令的 Gas 消耗等知识. 然而, 开发者的水平参差不齐、缺乏开发和优化智能合约的经验、对 Gas 机制的了解不够充足等原因, 导致大量低效的智能合约被部署到区块链[11].

3)编译器对 Gas 优化的支持非常有限. 如 1.1 节所述, 智能合约以字节码的形式部署和运行, 因此编译器有机会在编译阶段优化智能合约. 例如, Solidity 编译器向开发者提供了一个编译选项(即 optimize), 开启该编译选项后, 编译器将尝试优化大型常量存储和调度历程[26]. 然而, 尽管随着编译器的迭代, 新版本的编译器在 Gas 优化方面相比旧版本已经有所改进, 其优化效果仍然有限, 智能合约中大部分低效的指令序列都无法被消除[11]. 例如, 2.1 节中的实验表明: 即便是最新的编译器(即 2022 年 9 月 9 日发布的 Solidity 0.8.17), 也无法将 ExpensiveSampleContract 的 Gas 成本优化至 CheapSampleContract 的水平.

3 研究现状与进展

本节主要介绍智能合约 Gas 优化相关研究的现状与进展情况. 本文通过分析现有的相关文献, 将当前智能合约 Gas 优化的研究总结为 3 类方法: 面向智能合约字节码的优化方法、面向智能合约源代码的优化方法以及为一般软件设计的通用优化方法. 其中, 为一般软件设计的通用优化方法不是专门为智能合约设计的, 但它仍然可以应用于智能合约 Gas 优化.

3.1 面向字节码的优化

智能合约无论由何种高级编程语言开发, 它们最终都将被编译为字节码[22]. 研究者发现编译器生成的字节码中可能存在一些指令序列消耗大量 Gas, 如果能用一段语义相同但 Gas 消耗更少的指令序列替换它们, 就能达到节省资金的目的. 因此, 如何定义、检测并替换低效的指令序列, 是此类方法的研究重点. 具体而言, 面向字节码的优化方法通常分为 3 个阶段: 1)基于开发经验或实证研究确定低效的指令序列的模式(或特征); 2)借助上述模式从智能合约的字节码中检测待替换的指令序列; 3)用更高效的指令序列替换原来的指令序列以完成优化. 若新的指令序列中所有指令的总 Gas 消耗值比优化前小, 则优化后的智能合约可以为调用者节约交易费用; 若新的字节码长度比优化前更短, 则可以为开发者节省部署智能合约的成本.

关于低效的指令序列, Chen 等人[23] 在其论文中给出了详细的定义. 首先, 指令的执行被定义为将一个函数作用于一个 EVM 状态(即程序计数器、内存、栈和存储等), 使其转移到另一个状态. 例如, 执行 ADD 指令会使程序计数器和栈发生改变[7]. 然后, 基于以上对指令执行的定义, 给出低效的指令序列(在其论文中称为“anti-pattern”)的定义: 对于一段指令序列 OS_1, 如果存在另一段指令序列 OS_2 在语义上与 OS_1 等效, 但消耗更少 Gas, 则 OS_1 是低效的指令序列. 其中, OS_1 与 OS_2 的语义等效是指给定一个 EVM 状态, 分别执行 OS_1 和 OS_2 后得到的 2 个新的 EVM 状态是相等的. 让 gd_1, ge_1, gd_2, ge_2 分别表示部署和执行 OS_1 和 OS_2 所需的 Gas 数量, 由于智能合约只需部署一次但可能被调用多次, 所以用 T 表示指令序列所在的智能合约预计被调用的次数, 如果 $(gd_1-gd_2)+(ge_1-ge_2)\times T>0$, 则 OS_2 比 OS_1 更高效.

Gasper[22] 是第一个用于检测智能合约字节码中低效指令序列的工具. 如表 4 所示, 文献 [22] 首先人工地确定了两大类(共 7 种)低效指令序列, 即与无用代码相关的和与循环相关的低效指令序列. 例如, 死代码(dead code)是一种典型的与无用代码相关的低效指令序列, 它表示无论调用智能合约的哪个接口、用任何参数来调用, 都不会被执行的代码片段.

Chen 等人[22]认为如果可以减少智能合约的大小(例如删除死代码、删除不必要的比较操作)或者减少智能合约的计算(例如将昂贵的指令移出循环),则可以降低开发者和用户的成本. 因此, Chen 等[22]开发了 Gasper, 它能够从智能合约字节码中自动地识别两类低效的指令序列; Gasper 直接对字节码而非源代码进行检测, 因为只有极少数的智能合约是开放源代码的. 输入一个智能合约, Gasper 首先对字节码进行反汇编并构建控制流图, 然后基于控制流图对智能合约进行符号执行以探索和分析所有的程序路径, 在此过程中检测预先定义的 7 种低效指令序列. 例如, 识别不透明断言(详见表 4): Gasper 对智能合约进行符号执行, 在执行到条件跳转时记录其分支(即 true 和 false), 若检测到某个条件跳转具有一条从未执行的分支, 则识别到不透明断言. 目前, Gasper 支持识别 3 种低效指令序列, 即死代码、不透明断言和循环中的昂贵指令. 使用 Gasper 分析 2016 年 11 月 5 日之前部署的全部智能合约, 结果显示 93.5%, 90.1% 和 80% 的智能合约分别受这 3 种低效指令序列影响[22].

Table 4 Patterns of Two Types of Inefficient Instruction Sequences

表 4 两类低效指令序列的模式

类别	模式	描述
无用代码相关	死代码	合约中永远不会被执行到的片段
	不透明断言	不执行就能确定结果为 true 或 false 的断言
循环相关	循环中的昂贵指令	循环中可能重复执行的 Gas 消耗大的指令
	循环的常量结果	循环的结果是可以在编译时就能计算出的常量
	循环融合	若干个循环, 可以融合为一个循环
	循环中的重复计算	在循环的每次迭代中产生相同结果的表达式
	循环中固定结果的比较	循环中的比较操作, 比较结果不依赖于迭代次数

此后, Gasper 被扩展为 GasReducer[23], 旨在检测智能合约字节码中更多种类的低效指令序列并自动地将其替换为高效代码. 文献 [23] 首先通过人工分析已部署智能合约的历史执行轨迹(即 execution traces)总结出 24 个低效字节码的模式. 例如, 〈swap(X), swap(X)〉就是一个典型的低效指令序列, swap(X) 指令的作用是交换栈中 2 个元素的位置, 而 2 个这样的指令连续执行会使栈维持原状. 因此, 直接删去该指令序列不会影响原本的语义, 从而节省 6 个单位的 Gas(每个 swap(X) 指令的 Gas 消耗是 3). 针对这 24 种模式, GasReducer 首先将智能合约的字节码反汇编为汇编代码, 然后检查并记录汇编代码中的低效指令序列的位置. 其次, GasReducer 在每个记录的位置上把低效指令序列替换为相应的高效指令序列. 最后, 重新构造组装代码, 输出优化后的智能合约. GasReducer 对截至 2017 年 6 月 10 日部署的所有智能合约的检测结果显示[23]: 存在超过 940 万个低效指令序列, 为了部署它们, 开发者浪费了超过 20 亿个单位的 Gas; 通过检查这些智能合约的历史执行轨迹, GasReducer 发现在用户调用智能合约时, 这些低效字节码序列总共浪费了 70 亿个单位的 Gas.GasReducer 与 Gasper 的差别在于: 1)虽然两者都研究智能合约字节码中的低效指令序列, 但 Gasper 确定的低效指令序列来自于高级结构(例如循环), 而 GasReducer 直接来源于字节码; 2)GasReducer 可以识别并优化 24 种模式, 而 Gasper 只能检测 3 种并且不支持优化; 3)GasReducer 评估了部署和执行智能合约具体浪费的 Gas 数量.

GasChecker[11]是 Gasper 的另一个扩展版本. 区别于 GasReducer, 它不优化智能合约, 而是提升 Gasper 的扩展性以支持检测数百万个智能合约中的低效指令序列. 为了实现此目标, 文献 [11] 提出了利用云计算平台使符号执行并行化的方法. 具体而言, 它根据 MapReduce 编程模型对符号执行进行了并行化, 并且提出了基于反馈的负载均衡策略来提高计算资源的利用率. 实验结果表明, GasChecker 具有较高的精度(假阳性率约为 2.5%). 使用 GasChecker 分析 1 500 个已部署的智能合约, 结果显示平均每个合约中有超过 70 段低效指令序列. 此外, 文献 [11] 还研究了不同版本的编译器在 Gas 优化方面的作用, 即使用最新或较新的编译器编译智能合约, 再使用 GasChecker 检测生成的字节码. 实验结果表明最新的编译器能够优化少量旧的编译器无法优化的低效指令序列, 但大部分低效指令序列还是被保留下来. 为了量化 Gas 优化的效果, 文献 [11] 优化 1 500 个智能合约中的 3 种低效指令序列并与优化前对比, 结果显示, 如果部署和调用优化后的智能合约, 总共可以节约相当于 1 520 美元的 Gas.

以上识别和优化低效指令序列的方法均依赖于人工预先定义的模式, 这可能导致完整性不足的缺陷, 即人工定义的模式不足以覆盖智能合约中所有的低效指令序列; 此外, 人工定义还存在耗时、容易出错等问题. 因此, 为了克服上述缺陷, 一些研究尝试将超级优化技术[27]应用于智能合约 Gas 优化. 超级优化技术在 1987 年被提出. 给定一段需要优化的

指令序列,超级优化技术通过穷举搜索找出所有语义相同的指令序列,然后从中找出最优的一个来替换原指令序列[27]. 然而,超级优化通常被认为过于耗时,除非是特殊情况,否则无法在软件开发期间使用. 研究者认为智能合约就属于可以使用超级优化的一种特殊情况,因为智能合约一旦被部署到区块链上就无法被修改,但花费额外的时间来优化可能被无数次调用的程序是值得的[28]. 利用超级优化技术,智能合约 Gas 优化可以转换为一个对计算量要求较大的穷举搜索问题,而不依赖于人工预先定义的模式.

Nagele 和 Schett[28] 提出了一个名为 ebso 的智能合约字节码超级优化器,这是超级优化技术在智能合约上的首次应用. ebso 使用约束求解来优化智能合约字节码,为此它首先提供来自 EVM 的相关信息的编码(例如 EVM 状态、EVM 指令的编码),然后为字节码找到多种不同的目标程序,并确定其中 Gas 消耗最少的一个. ebso 对字节码中每个代码块(即一段不包含跳转指令的指令序列)执行上述优化,跨越不同代码块的指令序列不被考虑在内. Nagele 和 Schett[28] 在评估 ebso 时把它应用于一个编程竞赛的智能合约数据集,该竞赛旨在挑战为给定的编程题目编写 Gas 消耗最少的字节码,ebso 在这个已经达到相当高优化水平的数据集中仍然能找到 19 个可以进一步优化的代码块. 然而,实验结果还证实了超级优化技术的极端计算要求,即 ebso 在几乎 82% 的案例中出现了超时,这说明使用超级优化技术为代码块寻找最优替代仍然具有挑战性.

文献 [26] 提出了一种新的基于超级优化的智能合约 Gas 优化方法. 首先,该方法将智能合约字节码作为输入,通过符号执行获得控制流图中每个代码块的栈功能规约(stack functional specification, SFS),即对执行代码块前的初始栈和执行代码块后的最终栈的功能描述. 其次,该方法使用 SMT 求解器来合成符合栈功能规约的指令序列. 为了提升求解器的性能,该方法提出了一种有效的编码,只考虑栈操作(例如 PUSH, DUP 和 SWAP 等指令)的语义,其他指令被视为未解释的函数. 最后,该方法通过添加软约束来编码指令的 Gas 消耗,并利用最新的 Max-MAT 优化器提供的功能来改进搜索,得到符合栈功能规约的并且 Gas 消耗最小的指令序列. 文献 [26] 在一款名为 syrup 的工具中实现了这种基于超级优化的 Gas 优化方法,将它应用于 128 个最常被调用的智能合约可以获得 0.59% 的 Gas 节省量. 此外,syrup 只在处理 8.64% 的代码块时产生超时,这远远优于 ebso 的 92.12% 超时率.

文献 [16] 在 syrup 的基础上采用更多简化规则丰富栈功能规约、制定不同的约束条件来提高性能、增加栈功能规约验证以保证优化后的智能合约维持原始功能,并将 syrup 扩展到 syrup2.0. syrup2.0 可以与编译器集成,从而在编译时优化智能合约. 此外,文献 [16] 进行了一系列实验以确定优化效果和编译时间的最佳权衡.

另一种常见的优化技术是窥孔优化[29]. 编译器通过应用一些代码替换规则并使用模式匹配来优化滑动窗口(即窥孔)内的代码. 然而,这些替换规则通常依赖于人类的专业知识和经验;列举规则既耗时又缺乏系统性,因而找到合理的优化规则是窥孔优化的一个挑战. 智能合约由于发展时间较短,还没有累积足够的替换规则,例如:Solidity 的编译器 solc 只有十余条替换规则,而较成熟的 LLVM 已经有超过 1 000 条规则[29]. 针对此问题,文献 [29] 提出一个全新的方法以实现自动填充智能合约窥孔优化器中需要的替换规则. 首先,对现有的代码库执行超级优化,找到代码库中低效的指令序列以及它们的优化版本.然后,从上一步的优化结果中提取底层模式,生成窥孔优化规则. 文献 [29] 根据上述方法,结合超级优化器 ebso 和规则生成器 sorg,实现了一个面向 EVM 字节码的窥孔优化规则生成器 ppltr. 利用以太坊中 250 个最常被调用的智能合约来评估 ppltr, ppltr 生成了 993 个替换规则,然后将这些规则应用于 1 000 个最常被调用的智能合约,结果显示可以节省 4.58% 的 Gas.

3.2 面向源代码的优化

面向字节码的优化方法是从字节码到字节码的转换方法,然而开发者通常使用高级语言而不是字节码编写智能合约,这导致面向字节码的优化方法对智能合约开发者来说是不透明的,即开发者无法直观地看到优化方法对智能合约进行了哪些修改. 由于智能合约一旦部署到区块链就无法被修改,并且智能合约通常控制或影响着用户的资产,所以优化的不透明性会困扰开发者和用户,开发者和用户一方面希望减少智能合约的部署和执行成本,另一方面不希望不透明的优化过程中发生预期之外的修改(例如:智能合约原本的功能被改变),因为即便很微小的变化也可能引起大量的损失. 研究者还注意到尽管智能合约字节码分析工具能够分析所有已部署的智能合约(因为智能合约的字节码是区块链上公开可获取的信息),但是它们无法利用源代码中的有用信息(例如函数名和事件名)[30]. 因此,对于拥有

智能合约源代码的开发者来说，能够快速定位源代码中问题的分析工具可能比字节码分析工具更有用. 此外，一些研究[31-34]还表明字节码分析工具的效率低于源代码分析工具.

基于以上考虑，一些研究者关注面向源代码的优化方法，致力于让优化过程变得透明. 与面向字节码的优化方法类似，面向源代码的优化方法主要也分为 3 个阶段：1）基于开发经验或实证研究确定可能引起 Gas 浪费的编程模式、代码片段的特征；2）借助模式匹配等方法从智能合约中识别编程模式或特征；3）用更高效的代码片段替换识别出的低效代码片段以完成优化.

Tikhomirov 等人[35]总结了 21 种智能合约的代码问题，其中 2 种代码问题与低效代码有关：第 1 种低效代码是被定义为 byte[] 类型的数据，它们可以被更节省 Gas 消耗的 bytes 类型替代；第 2 种低效代码是内部带有函数调用的循环，因为重复的函数调用将导致相当大的 Gas 消耗. 值得注意的是，第 1 种低效代码很难被面向字节码的优化方法识别，因为字节码中不包含类型信息，导致其很难在不借助源代码的情况下区分 byte[] 和 bytes. Tikhomirov 等人[35]又开发了一个名为 SmartCheck 的可扩展静态分析工具以检测这些代码问题. SmartCheck 首先对 Solidity 源代码进行词法和句法分析，将源代码转换为基于 XML 解析树的中间表示. 然后，SmartCheck 对该中间表示使用 Xpath 查询以检测预定义的代码问题. 文献 [35] 对大量现实世界的智能合约进行测试，检测到 0.006% 的智能合约具有第 1 种低效代码，2.164% 的智能合约具有第 2 种低效代码.

Zhang 等人[30]提出 SolidityCheck，旨在使用正则表达式定义智能合约中存在问题的语句特征，并使用正则匹配来检测对应的问题. 关注 SmartCheck 中总结出的两类与低效代码相关的代码问题，实验结果表明，使用正则匹配来识别代码问题比使用词法和句法分析更高效. SolidityCheck 将分析单个合约所消耗的时间从 SmartCheck 的 1.83 s 缩短到了 0.23 s.

Feist 等人[36]提出的智能合约静态分析框架 Slither 可以检测 2 种导致智能合约部署和执行成本变高的代码模式：第 1 种是应该声明为常量却被声明为变量的数据，如果变量被声明为常量，数据将不会占用智能合约的存储空间，并且在使用数据时可以执行更少的指令，Slither 是唯一一个能找到此类代码模式的可用工具；第 2 种是本应该声明为外部函数却未被声明为外部函数的函数，因为外部函数的代码可以被编译器优化.Slither 将 Solidity 代码转化为一种中间表示，然后使用数据流和污点分析从中间表示中提取这 2 种代码模式，并对 Slither 可以检测的第 1 种低效代码模式进行了评估，结果显示超过 50% 的智能合约至少包含一个应该声明为常量的变量，其中甚至包含一些从未被访问过的变量.

以上面向智能合约源代码的研究[30,35-36]根据预先定义的代码模式从源代码中标记出低效的代码片段，从而帮助开发者改进智能合约. 此后在此基础上进一步研究针对低效代码片段的自动优化.

据我们所知，GASOL[37]是第一款支持自动检测并优化智能合约源代码中低效代码片段的工具. GASOL 的优化目标是减少与访问存储（即 SSTORE 指令）相关的 Gas 消耗. GASOL 包含 1 个分析模块和 1 个优化模块，如果分析模块检测到智能合约中对相同存储位置的多次访问，则 GASOL 将其视为一个潜在的优化机会，优化模块将尝试使用节省 Gas 的内存访问来代替存储访问. 具体而言，GASOL 的目标是用 1 次访问来替换 1 个循环结构中对同一个全局变量的多次访问（如表 2 所示，存储操作的 Gas 消耗为 5 000 或 20 000），它将存储空间中的数据复制到内存中，然后访问该内存地址（如表 2 所示，内存操作的 Gas 消耗仅为 3），并且只在必要时对存储进行最终更新. GASOL 将这种对存储访问的优化方法应用于超过 40 000 个真实智能合约的公共函数，结果显示 6.81% 的公共函数都存在优化机会[38].

同样针对智能合约中的存储操作，文献 [39] 注意到现有工作没有考虑到的 2 个方面：1）控制流结构不仅包含循环，还包含各种分支，以前的研究工作[11, 37]在优化智能合约时只考虑了循环结构中的存储操作. 因此，以前的研究工作[11, 37]在减少用于存储访问的 Gas 消耗时可能导致其他程序路径上的 Gas 消耗增加. 2）文献 [36-37] 都只考虑基本数据类型（例如 int, bool）而没有考虑对复合数据类型的优化（例如数组），然而 90% 的包含循环的智能合约都至少在一个函数中使用了数组，这意味着在数组访问方面存在较大的 Gas 优化机会. 文献 [39] 针对上述 2 个方面提出解决方案以弥补现有工作的不足：1）针对存储访问的优化，优化目标是：给定一个智能合约 C，将其优化为 C'，使得对于 C 中的每个函数 f，C'都有一个对应的函数 f'，并且对于 f 中的每个执行路径 p，f'都有一个与其语义相同的执行路径 p'，其中 p'访问存储的次数比 p 少.文献 [39] 提出的优化方法是在 GASOL 的基础上增加了对每条程序路径的优化而不仅限于

循环结构. 2)针对数组访问的优化，其目标是消除不必要的数组越界检查所带来的 Gas 消耗. 根据 Solidity 语言规范，每个数组访问都需要进行运行时的越界检查以避免非法越界，保证用于访问数组的索引不超过声明的数组大小(0<i<l−1, i: 索引; l: 数组长度). 为了实现越界检查，字节码中需要比较索引和数组长度，这在某些情况下是不必要的，例如在编译时就能确定索引一定小于数组长度的情况. 此外，对于动态长度的数组，其长度存放在存储空间中，导致在执行越界检查时需要从存储中读取数组长度. 如果在多次访问期间数组长度始终不变，则可以通过一个局部变量存放数组长度，然后在越界检查时从局部变量中读取数组长度，而不用访问存储空间，达到节省 Gas 的目的. 为了优化数组越界检查和获取动态数组的长度，文献 [39] 建议使用功能等效但 Gas 消耗更少的内联汇编来访问数组.

Liao 等人[21] 发现内联汇编还可以在 5 种情况下替代 Solidity 源代码以优化智能合约的 Gas 消耗. 本节只介绍 2 种: 1)类型转换. Solidity 的类型系统限制了每种类型的变量可用的指令，比如算数运算的操作数不能是一个地址类型的变量. 若试图对地址类型变量进行算数运算，必须提前将地址类型变量转换为整数类型. 然而，类型转换需要执行特定的指令，从而消耗更多 Gas. 使用内联汇编可以解决此问题，因为内联汇编中没有类型的区分，所以地址类型变量在内联汇编中无需转换就可以进行算数运算. 2)访问字符串的任意长度子串. Solidity 不支持直接访问字符串的子串或字符串中的某个字符，若要实现此功能，智能合约必须先将字符串转换为 bytes 类型，然后再用一个循环来获取其子串. 内联汇编可以优化此过程，这是因为内联汇编允许开发者直接使用 MLOAD 指令来访问内存的任意位置，从而获取字符串的子串. 使用内联汇编以优化智能合约的其余 3 种情况请参阅文献 [21] 和表 5.

回顾本节列举出的面向源代码的优化方法，我们发现，研究者通常提出一些低效的代码模式，然后运用正则匹配、静态程序分析等方法从源代码中识别并优化此类低效代码. 然而，大多数研究没有阐述低效代码模式是如何获得的(例如: 通过人工分析真实的已部署的智能合约来获得，或从随机构造的代码中获得)，这可能导致开发者和研究人员无法了解这些代码模式在真实智能合约中的泛滥程度. 即便有研究通过实验评估了一些低效代码模式的占有率[35-37]，文献 [35-37] 所覆盖的少量代码模式也无法揭示宏观情况.

Table 5 Saving Gas by Using Inline Assembly

表 5 使用内联汇编节省 Gas

模式	描述
do…while	使用内联汇编提供的 do…while 结构替换 Solidity 提供的 for 和 while 循环（注意：仅在 Solidity 0.4.5 版本前有效）.
数组越界检查	使用内联汇编访问数组，以避免不必要的越界检查.
访问子串	使用内联汇编可以直接访问字符串的任意子串或其中单个字符，而 Solidity 代码必须借助更复杂的操作.
按位移位	Solidity 代码中编写的按位移位被编译为乘和除指令，比内联汇编中的移位指令的 Gas 消耗更高（注意：仅在 Solidity 0.5.5 版本前有效）.
智能合约调用	若智能合约用内联汇编调用另一个智能合约，将比使用 Solidity 代码节省 Gas.

对开发者和真实智能合约进行大规模的调查，总结开发者面临的智能合约优化问题和真实智能合约中实际存在的低效代码模式，是解决开发者和研究人员无法了解低效代码模式在真实智能合约中的泛滥程度的问题的一个方法. 例如，Kong 等人[40] 从开发者论坛的大量帖子中总结低效代码模式，并通过大规模的实证研究揭示这些模式的普遍性. 据悉，在线论坛和问答网站(例如：由 Solidity 开发团队推荐的官方讨论网站——Ethereum Stack Exchange[41])中包含的短文、问题和答案等非正式文档是智能合约开发者的宝贵资源，其中可以找到低效智能合约的真实案例以及关于如何进行 Gas 优化的建议. 因此，文献 [40] 从 Ethereum Stack Exchange 的帖子中提取智能合约的低效代码模式.具体而言，首先爬取该网站截至 2020 年 7 月 9 日的所有 25 000 个帖子(包含问题、答案和评论)；然后利用关键词过滤、手动过滤与 Gas 优化不相关的帖子；进一步，从剩余的 474 个帖子中总结出 6 种低效代码模式以及对应的优化方法[40]，其中只有“重复赋值”和“频繁使用全局变量”这 2 种模式被研究过. 基于发现的低效代码模式，文献 [40] 提出了一种 Gas 优化方法，该方法利用抽象语法树从源代码级别优化智能合约，并构造源代码到源代码的转换，旨在让优化过程对开发者透明；最后，将提出的检测和优化方法应用于包含超过 160 000 个真实智能合约的数据集上，实验结果表明，52.75% 的合约至少包含 1 种低效代码模式，优化这些低效代码模式将至少为每个智能合约减少 0.3 美元的成本.

区别于其他面向源代码的优化方法，Chen 等人[42] 从数据类型转换的角度而非替换低效代码的角度重构智能合约以节省 Gas. 文献 [42] 提出，优化智能合约的 Gas 消耗需要对底层数据布局进行显著地修改，这要求开发者尝试更改不同的数据结构、重新实现

智能合约中某些重要的代码段，而以往的所有研究都没有解决此问题. 基于此观察，文献 [42] 开发了一个名为 Solidare 的工具，用于自动执行智能合约的数据结构重构任务. 给定智能合约的 Solidity 源代码和所需的数据类型重构操作，Solidare 将自动生成一个带有重构数据类型的等效智能合约. 借助 Solidare，智能合约开发者可以方便、快速地尝试各种不同的数据表示并测量重新实现的智能合约的 Gas 消耗量. 将 Solidare 应用到 20 个真实的智能合约上，其中 18 个智能合约的 Gas 消耗量平均减少了 16%.

3.3 通用优化方法

智能合约是一种特殊的计算机软件，因此，软件工程领域的优化方法也能被运用到智能合约的开发过程中.

例如，文献 [43] 基于对 Solidity 文档、博客和开发者论坛等在线资源和现有智能合约的调查，提出了 5 类共 14 种可以帮助开发者进行智能合约 Gas 优化的设计模式，每个设计模式是针对某一个频繁出现的设计问题的典型解决方案. 这 5 类设计模式被概括为：1）外部交易. 例如尽量让外部系统直接访问区块链中的事件日志而不要将信息存放在智能合约的存储空间中；2）存储. 例如始终使用内存来存储非持久化数据；3）节约空间. 例如建议使用 mapping 类型来管理数据列表；4）函数功能. 例如使用库；5）其他. 例如借助编译器的优化选项.

文献 [44] 将智能合约在结构和功能方面的优化归结为 2 个基本层面，即使用适当类型的变量和使用最优方式实现智能合约的功能. 文献 [44] 主要研究如何为可能产生大量 Gas 消耗的元素（例如循环、数据类型）编写最优的智能合约源代码. 更重要的是，文献 [44] 提出了一系列建议开发者在开发智能合约时遵循的有序优化步骤. 文献 [11, 21-23, 26, 28-30, 35, 37, 39-40, 42] 虽然提出了很多优化建议和方法，但没有指导开发者以何种次序采用不同的优化方法取得最好的效果. 为了弥补这个空白，文献 [44] 提出的指导原则包含 11 个建议：1）为智能合约创建适当的数据类型；2）变量有时需要重新排序或打包，以充分利用存储空间；3）尽量使用内存而不是存储中的变量；4）尽可能使用关键字 constant 声明智能合约中的常量；5）使用更节省 Gas 的 mapping 类型来替代数组类型；6）特别注意优化循环相关的这部分源代码，例如尽量不在循环中添加 Gas 消耗高的指令；7）最好调用内部函数；8）使用 OR 或 AND 运算符时，由于“短路”[44]，运算符的右操作数可能被跳过，因此建议将 Gas 消耗高的操作放在运算符的右边；9）消除无用代码；10）消除不必要的变量；11）开启编译器优化.

Bentley[45] 曾在 1982 年为一般软件制定了一套通用和抽象的优化规则.近年来，研究者发现这些规则可能有助于优化智能合约[46].具体来说，这套规则分为 6 类：1）空间换时间规则. 通过增加所需的空间以减少程序的运行时间来实现优化；2）时间换空间规则. 存储冗余信息以减少程序运行时间；3）循环规则. 通过修改或删除循环来实现优化；4）逻辑规则. 旨在不改变语义的情况下修改代码逻辑以提高效率；5）步骤规则. 该规则处理的是一个按步骤组织的程序的底层结构；6）表达式规则. 处理表达式优化，例如复用计算结果或用开销小的替换开销大的表达式.

3.4 不同类型优化方法的对比

目前，智能合约 Gas 优化方法主要分为 3 类，即面向字节码的优化方法、面向源代码的优化方法和通用优化方法，它们具有各自的优势和劣势.

回顾 1.1 节中描述的智能合约生命周期，即无论智能合约最初由何种高级语言编写，在部署上链前都须编译为 EVM 字节码. 因此，面向字节码的优化方法能够兼容不同编程语言编写的智能合约. 然而，此类方法通过字节码到字节码的转换实现 Gas 优化，开发者更容易阅读、理解源代码，这导致开发者难以判断优化过程中是否发生了预期之外的错误.

为了让优化过程对开发者更透明，一些研究工作关注面向源代码的优化方法. 通过对比优化前后的智能合约源代码，开发者能够容易地定位到优化的位置，并利用自身开发经验判断优化是否合理. 因此，面向源代码的优化方法对开发者更友好、透明. 此外，由于只有源代码中包含类型信息，所以此方法还能识别到一些与变量类型相关的优化机会. 然而，此类优化方法依赖于特定的编程语言，不同编程语言编写的智能合约源代码具有显著差异，因而针对一种语言的优化方法往往无法快速移植到另一种语言上.

相比于面向字节码和面向源代码的优化方法，通用优化方法依赖于软件工程领域的经验与方法论，而无需设计、开发专门的优化工具或程序，因此相对容易实施. 然而，优化效果的好坏取决于开发者是否积累了足够的经验，所以此类方法对开发者的水平提出了较高要求. 表 6 从 3 种不同类型的方法中挑选出了代表性的优化技术，并进行了对比.

3.5 与 Gas 相关的其他研究

除了围绕智能合约 Gas 优化的研究工作，还有研

究者关注 Gas 的其他方面,例如.估计执行智能合约所需的 Gas、识别智能合约中与 Gas 相关的漏洞等.本节简要介绍这些研究工作.

V-Gas[47] 利用静态分析和模糊测试方法对执行智能合约所需的最大 Gas 消耗量进行估计,并以此为智能合约用户提供设置 Gas 总限额的建议,从而避免因 Gas 总限额设置太小而引起的 Gas 耗尽异常. GASTAP[48] 是一个 Gas 感知(Gas-Aware)智能合约分析平台,它将智能合约作为输入,并通过构建控制流图、从低级代码到高级表示的反编译等一系列代码转换和分析,自动推断该合约中所有公共(public)函数的 Gas 消耗量上限. Marescotti 等人[49] 提出了一种受有界模型检查技术启发的技术,并基于该技术计算智能合约最坏情况下的准确 Gas 消耗量. Soto 等人[50] 通过搭建私有区块链并随机生成交易来模拟公共区块链,他们在私有区块链中多次测试执行智能合约的 Gas 消耗量,以此帮助用户预测在公共区块链中调用智能合约所需的成本. Li 等人[51] 认为静态分析难以确定智能合约中循环体的执行次数,因此静态分析方法无法准确估计循环结构的 Gas 消耗量.针对此问题,他们提出了一种基于交易轨迹的 Gas 估计方法,该方法通过了解智能合约的过往交易所消耗的 Gas 量来估计新交易的 Gas 消耗.

Li 等人[52] 提出了一种动态数据复制方案 GRuB,可在区块链和链外的云存储之间动态地复制数据,从而降低数据密集型智能合约的 Gas 消耗.

Grech 等人[25] 对与 Gas 相关的合约漏洞进行了分类总结,他们认为这类漏洞是开发者最难防范的漏洞之一,因为与 Gas 相关的合约漏洞在非攻击场景中很难被触发.因此,他们提出了一种高精度、可扩展的静态程序分析技术 MadMax,用于自动检测与 Gas 相关的漏洞.

Chen 等人[53] 提出了一个基于仿真的框架来自动测量执行每个 EVM 指令所产生的计算资源消耗量.基于该框架,他们发现目前为每个指令制定的固定 Gas 消耗值不能适应指令在不同环境下的资源消耗量的变化.因此,攻击者可以利用 Gas 消耗较小而计算资源开销大的指令发动拒绝服务攻击.针对此问题,他们提出了一种新的 Gas 消耗机制,该机制根据指令的执行次数动态地调整指令的 Gas 消耗值,从而阻止拒绝服务攻击.

4 未来研究方向

区块链是一项新兴的技术,其发展速度可谓日新月异.近年来,基于区块链的智能合约被认为是一个极具潜力和前景的领域.随着研究者和开发者将智能合约的使用场景拓展到包括供应链[54]、物联网[55]、医疗系统[56]、数字版权管理[57]、保险[58]、金融体系[59]和房地产[60] 在内的各种领域,智能合约的数量与日俱增.智能合约的繁荣应用吸引了大量的用户,例如,以太坊区块链上每天可以产生超过一百万条交易[61],用于支付这些交易的费用价值高达数百万美元[24].总而言之,随着智能合约及其应用的发展,可预见越来越多的智能合约将被部署到区块链、越来越多的用户将使用智能合约.因此,为减少资源浪费并保证区块链健康持续发展,对智能合约 Gas 优化的重要性的认识正不断提升,需要继续对其进行研究以弥补现有优化方法的缺陷和不足.本文将未来研究方向归纳为以下 4 个方面.

1)增加对更多智能合约编程语言的研究.据我们所知,现有面向源代码的优化方法都以 Solidity 语言编写的智能合约为优化对象,然而其他的智能合约编程语言(例如 Vyper[62])没有被研究过.使用这些编程语言的开发者不能从现有优化方法中获得支持,导致开发出浪费 Gas 的低效智能合约.针对此问题,文献 [36] 提出,可以先将其他语言编写的智能合约转化为现有的中间表示,然后对该中间表示实施优

Table 6 Comparison of Different Types of Optimization Methods

表 6 不同类型优化方法的对比

类型	优化技术	优点	缺点
面向字节码的优化	基于预定义模式	执行效率高、兼容不同编程语言	完整性不足、需要人工、对开发者不透明
	超级优化	优化效果好、兼容不同编程语言	穷举搜索过于耗时、对开发者不透明
	窥孔优化	可在编译时自动优化、兼容不同编程语言	目前没有足够的替换规则、对开发者不透明
面向源代码的优化	基于预定义模式	执行效率高、开发者友好、可以利用源代码信息	完整性不足、需要人工、不兼容不同编程语言
	数据类型转换	对特定智能合约效果很好、可以利用源代码信息	不兼容不同编程语言、仅针对特定智能合约
通用优化		无需专门的优化程序	依赖软件工程方法论及经验

化. 除了文献 [36] 提出的方法，面向字节码的优化方法也能在一定程度上解决该问题，因为无论何种语言编写的智能合约，在部署到区块链之前都需要编译成 EVM 字节码. 但是，面向字节码的优化方法存在不透明性，有待在未来的研究工作中解决. 此外，现有面向字节码的优化方法没有评估其对不同语言编写的智能合约的优化效果.

2）提升超级优化技术的扩展性和效率. 超级优化技术因为过于耗时而阻碍了其广泛应用，虽然现有研究将其应用到智能合约上并得到了 Gas 优化效果，但是超级优化技术存在的固有缺陷影响了 Gas 优化效果和效率. 例如：①现有智能合约超级优化方法对单个代码块内的指令序列进行优化，而无法实现跨代码块的优化；②由于超级优化依赖大量计算，为了减少时空开销，现有研究只对栈操作编码，而不考虑其他指令（例如内存、存储操作），导致优化范围有限. 未来研究或许可以借鉴 GasChecker 的思路，即利用云计算平台使一些操作并行化，从而提升超级优化的效率.

3）将优化方法集成到智能合约编译器. 目前，编译器对 Gas 优化的支持非常有限，文献 [11] 发现即便开启编译器的优化选项，仍然无法消除绝大部分低效指令序列. 如 2.3 节所述，即便是最新的编译器，也无法优化图 1（a）中展示的低效代码. 值得注意的是，这种低效代码模式已经能够被现有工具 GASOL 识别并优化. 如果在未来的研究工作中能将现有的优化方法和工具集成到智能合约编译器中，这将为开发者提供极大的便利. 据我们所知，syrup2.0 已经在这方面做出了努力，但是超级优化技术的固有缺陷限制了它的优化效果和效率.

4）设计一种在智能合约运行时进行 Gas 优化的方法. 智能合约一旦部署到区块链就无法被修改，因此现有的基于代码替换的 Gas 优化方法无法应用于已部署的智能合约. 这部分智能合约可能在未来还会被多次调用，从而产生难以避免的资源浪费. 未来的研究工作或许可以考虑在智能合约运行时进行 Gas 优化，以减少实际执行的指令数量，从而在不修改已部署智能合约代码的情况下减少 Gas 消耗量.

5 总　　结

基于区块链的智能合约是目前广受研究者和开发者关注的技术，未经优化的智能合约将浪费 Gas 和计算资源.本文总结目前智能合约 Gas 优化的研究进展：1）介绍了 Gas 机制的基本概念；2）阐述了智能合约 Gas 优化的动机；3）总结了实施优化所面临的 3 项主要挑战；4）总结了当前智能合约 Gas 优化研究的 3 大方向，即面向字节码的方法、面向源代码的方法和通用优化方法；5）从现有研究的不足出发，展望了未来研究的发展方向.

作者贡献声明：宋书玮负责收集文献并撰写论文；倪孝泽负责完成实验并修改论文；陈厅提出指导意见并修改论文.

参　考　文　献

[1] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system [EB/OL]. (2008−10−31) [2022−09−14].https://bitcoin.org/bitcoin.pdf

[2] von Haller Gronbaek M. Blockchain 2.0, smart contracts and challenges [EB/OL]. [2022−09−14].https://www.twobirds.com/-/media/pdfs/in-focus/fintech/blockchain2_0_martinvonhallergroenbaek_08_06_16.pdf

[3] Zou Weiqin, Lo D, Kochhar P, et al. Smart contract development: Challenges and opportunities[J]. IEEE Transactions on Software Engineering, 2019, 47(10): 2084−2106

[4] Ruth A. Why build decentralized applications: Understanding Dapp [EB/OL]. (2022−01−17) [2022−09−14].https://due.com/blog/why-build-decentralized-applications-understanding-dapp/

[5] Ethereum. org. Ethereum [EB/OL]. [2022−09−14].https://ethereum.org/en/

[6] Young M. Over 44 million contracts deployed to Ethereum since genesis: Research [EB/OL]. (2022−08−04) [2022−09−14].https://cryptopotato.com/over-44-million-contracts-deployed-to-ethereum-since-genesis-research/

[7] Wood G. Ethereum: A secure decentralised generalised transaction ledger[J]. Ethereum Project Yellow Paper, 2014, 151(2014): 1−32

[8] Chen Ting, Li Xiaoqi, Wang Ying, et al. An adaptive gas cost mechanism for Ethereum to defend against under-priced DoS attacks [J]. arXiv preprint, arXiv: 1712.06438, 2017

[9] Binance. An Introduction to BNB Smart Chain (BSC) [EB/OL]. [2022−10−09].https://academy.binance.com/en/articles/an-introduction-to-binance-smart-chain-bsc

[10] Polygon technology. Polygon [EB/OL]. [2022−10−09]. https://polygon.technology/

[11] Chen Ting, Feng Youzheng, Li Zihao, et al. Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts[J]. IEEE Transactions on Emerging Topics in Computing, 2020, 9(3): 1433−1448

[12] Monrat A A, Schelén O, Andersson K. A survey of blockchain from the perspectives of applications, challenges, and opportunities[J]. IEEE Access, 2019, 7(1): 117134−117151

[13] Szabo N. Formalizing and securing relationships on public networks[J]. First Monday, 1997, 2(9): 1−22

[14] Clack C D, McGonagle C. Smart derivatives contracts: The ISDA master agreement and the automation of payments and deliveries [J]. arXiv preprint, arXiv: 1904.01461, 2019

[15] Wang Zeli, Jin Hai, Dai Weiqi, et al. Ethereum smart contract security research: Survey and future research opportunities[J]. Frontiers of Computer Science, 2021, 15(2): 1−18

[16] Albert E, Gordillo P, Hernández-Cerezo A, et al. Super-optimization of smart contracts [J]. ACM Transactions on Software Engineering and Methodology, 2022, 31(4): Article 70

[17] Liu Chao, Gao Jianbo, Li Yue, et al. Understanding out of gas exceptions on Ethereum [C] //Proc of Int Conf on Blockchain and Trustworthy Systems. Berlin: Springer, 2019: 505 − 519

[18] Albert E, Correas J, Gordillo P, et al. GASOL: Gas Analysis and optimization for Ethereum smart contracts [J]. arXiv preprint, arXiv: 1912.11929, 2019

[19] Etherscan. Ethereum average gas price chart [EB/OL]. (2022−09−23) [2022−09−26].https://etherscan.io/chart/gasprice

[20] Etherscan. Ether daily price (USD) chart [EB/OL]. (2022−09−23) [2022−09−26].https://etherscan.io/chart/etherprice

[21] Liao Zhou, Song Shuwei, Zhu Hang, et al. Large-Scale Empirical Study of Inline Assembly on 7.6 Million Ethereum Smart Contracts [J]. IEEE Transactions on Software Engineering, 2022, Early Access: 1 − 25

[22] Chen Ting, Li Xiaoqi, Luo Xiapu. Under-optimized smart contracts devour your money [C] //Proc of the IEEE 24th Int Conf on Software Analysis, Evolution and Reengineering (SANER). Piscataway, NJ: IEEE, 2017: 442 − 446

[23] Chen Ting, Li Zihao, Zhou Hao, et al. Towards saving money in using smart contracts [C] //Proc of the IEEE/ACM 40th Int Conf on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER). Piscataway, NJ: IEEE, 2018: 81 − 84

[24] Brandstätter T. Optimization of solidity smart contracts [D]. Vienna: Vienna University of Technology, 2020

[25] 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

[26] Albert E, Gordillo P, Rubio A, et al. Synthesis of super-optimized smart contracts using max-smt [C] //proc of Int Conf on Computer Aided Verification. Berlin: Springer, 2020: 177 − 200

[27] Massalin H. Superoptimizer: A look at the smallest program[J]. ACM SIGARCH Computer Architecture News, 1987, 15(5): 122−126

[28] Nagele J, Schett M A. Blockchain superoptimizer [J]. arXiv preprint, arXiv: 2005.05912, 2020

[29] Schett M A, Nagele J. Populating the Peephole Optimizer of a Smart Contract Compiler [C] //Proc of the 2nd Workshop on Formal Methods for Blockchains (FMBC). Wadern: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020: 1 − 15

[30] Zhang Pengcheng, Xiao Feng, Luo Xiapu. SolidityCheck: Quickly detecting smart contract problems through regular expressions [J]. arXiv preprint, arXiv: 1911.09425, 2019

[31] Luu L, Chu D, Olickel H, et al. Making smart contracts smarter [C] //Proc of 2016 ACM SIGSAC Conf on Computer and Communications security. New York: ACM, 2016: 254 − 269

[32] Nikolić I, Kolluri A, Sergey I, et al. Finding the greedy, prodigal, and suicidal contracts at scale [C] //Proc of the 34th Annual Computer Security Applications Conf. New York: ACM, 2018: 653 − 663

[33] Torres C F, Schütte J, State R. Osiris: Hunting for integer bugs in ethereum smart contracts [C] //Proc of the 34th Annual Computer Security Applications Conf. New York: ACM, 2018: 664 − 676

[34] Tsankov P, Dan A, Drachsler-Cohen D, et al. Securify: Practical security analysis of smart contracts [C] //Proc of the 2018 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 67 − 82

[35] Tikhomirov S, Voskresenskaya E, Ivanitskiy I, et al. SmartCheck: Static analysis of Ethereum smart contracts [C] //Proc of IEEE/ACM 1st Int Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). New York: ACM, 2018: 9 − 16

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

[37] Albert E, Correas J, Gordillo P, et al. GASOL: Gas analysis and optimization for Ethereum smart contracts [C] //Proc of Int Conf on Tools and Algorithms for the Construction and Analysis of Systems. Berlin: Springer, 2020: 118 − 125

[38] Correas J, Gordillo P, Román-Díez G. Static profiling and optimization of Ethereum smart contracts using resource Analysis[J]. IEEE Access, 2021, 9(1): 25495−25507

[39] Li Chunmiao. Gas estimation and optimization for smart contracts on Ethereum [C] //Proc of the 36th IEEE/ACM Int Conf on Automated Software Engineering (ASE). Piscataway, NJ: IEEE, 2021: 1082 − 1086

[40] Kong Queping, Wang Ziyan, Huang Yuan, et al. Characterizing and detecting gas-inefficient patterns in smart contracts[J]. Journal of Computer Science and Technology, 2022, 37(1): 67−82

[41] Inc Stack Exchange. Ethereum stack exchange [EB/OL]. [2022−10−08].https://ethereum.stackexchange.com/

[42] Chen Yanju, Wang Yuepeng, Goyal M, et al. Synthesis-powered optimization of smart contracts via data type refactoring[J]. Proceedings of the ACM on Programming Languages, 2022, 6(OOPSLA2): 560−588

[43] Marchesi L, Marchesi M, Destefanis G, et al. Design patterns for gas optimization in ethereum [C] //Proc of IEEE Int Workshop on Blockchain Oriented Software Engineering (IWBOSE). Piscataway, NJ: IEEE, 2020: 9 − 15

[44] Čeke D, Kunosić S, Buzađija N. Smart contract execution costs optimisation on blockchain network [C] //Proc of the 45th Jubilee Int Convention on Information, Communication and Electronic Technology (MIPRO). Piscataway, NJ: IEEE, 2022: 1442 − 1447

[45] Bentley J L. Writing Efficient Programs [M]. Upper Saddle River:

Prentice-Hall, Inc. , 1982

[46] Brandstätter T, Schulte S, Cito J, et al. Characterizing efficiency optimizations in solidity smart contracts [C] //Proc of IEEE Int Conf on Blockchain (Blockchain). Piscataway, NJ: IEEE, 2020: 281 − 290

[47] Ma Fuchen, Ren Meng, Ying Fu, et al. V-Gas: Generating high gas consumption inputs to avoid out-of-Gas vulnerability [J/OL]. ACM Transactions on Internet Technology (TOIT), 2018. https://dl.acm.org/doi/abs/10.1145/3511900?casa_token=vx-wRTMrwlkAAAAA:_Z1FzwVEFB9C7EnHGiVQTeMmhuxqqt6g2Cxjp6OXXMZwvDJXe36pyOcVPJavIlxh8gVeXXGXqQHgz8o

[48] Albert E, Gordillo P, Rubio A, et al. GASTAP: A gas analyzer for smart contracts [J]. arXiv preprint, arXiv: 1811.10403, 2018

[49] Marescotti M, Blicha M, Hyvärinen A E J, et al. Computing exact worst-case gas consumption for smart contracts [C] //Proc of Int Symp on Leveraging Applications of Formal Methods. Berlin: Springer, 2018: 450 − 465

[50] Soto D, Bergel A, Hevia A. Fuzzing to estimate gas costs of Ethereum contracts [C] //Proc of IEEE Int Conf on Software Maintenance and Evolution (ICSME). Piscataway, NJ: IEEE, 2020: 687 − 691

[51] Li Chunmiao, Nie Shijie, Cao Yang, et al. Trace-based dynamic gas estimation of loops in smart contracts[J]. IEEE Open Journal of the Computer Society, 2020, 1(1): 295−306

[52] Chen Jiaqi, Li Kai, Yu Zhe, et al. GRuB: Gas-efficient blockchain storage via workload-adaptive data replication [J]. arXiv preprint, arXiv: 1911.04078, 2019

[53] Chen Ting, Li Xiaoqi, Wang Ying, et al. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks [C] //Proc of Int conf on Information Security Practice and Experience. Berlin: Springer, 2017: 3 − 24

[54] Korpela K, Hallikas J, Dahlberg T. Digital supply chain transformation toward blockchain integration [C] //Proc of the 50th Hawaii Int Conf on System Sciences. Mānoa: ScholarSpace, 2017: 1 − 10

[55] Reyna A, Martín C, Chen J, et al. On blockchain and its integration with IoT. Challenges and opportunities[J]. Future Generation Computer Systems, 2018, 88(1): 173−190

[56] Chakraborty S, Aich S, Kim H. A secure healthcare system design framework using blockchain technology [C] //Proc of 21st Int Conf on Advanced Communication Technology (ICACT). Piscataway, NJ: IEEE, 2019: 260 − 264

[57] Ma Zhaofeng, Jiang Ming, Gao Hongmin, et al. Blockchain for digital rights management[J]. Future Generation Computer Systems, 2018, 89(1): 746−764

[58] Gatteschi V, Lamberti F, Demartini C, et al. Blockchain and smart contracts for insurance: Is the technology mature enough? [J]. Future Internet, 2018, 10(2): 20

[59] Fanning K, Centers D P. Blockchain and its coming impact on financial services[J]. Journal of Corporate Accounting & Finance, 2016, 27(5): 53−57

[60] Karamitsos I, Papadaki M, Al Barghuthi N B. Design of the blockchain smart contract: A use case for real estate[J]. Journal of Information Security, 2018, 9(3): 177−177

[61] Etherscan. Ethereum daily transactions chart [EB/OL]. [2022−09−26].https://etherscan.io/chart/tx

[62] Vyperlang. Vyper [EB/OL]. [2022−10−08].https://github.com/vyperlang/vyper

Song Shuwei, born in 1999. PhD candidate. His main research interests include blockchain security and software security.

宋书玮, 1999 年生. 博士研究生.主要研究方向为区块链安全和软件安全.

Ni Xiaoze, born in 1999. Master candidate. His main research interests include blockchain security and software security.

倪孝泽, 1999 年生. 硕士研究生.主要研究方向为区块链安全和软件安全.

Chen Ting, born in 1987. PhD, professor. Member of CCF. His main research interests include blockchain security, software security and software engineering.

陈　厅, 1987 年生. 博士, 教授. CCF 会员.主要研究方向为区块链安全, 软件安全和软件工程.