arXiv:2301.04757v1 [cs.PL] 11 Jan 2023

# 推断以太坊字节码中的不必要内存写入访问（扩展版）*

Elvira Albert[1], Jes´ús Correas[1], Pablo Gordillo[1](✉), Guillermo Rom´án-D´íez[2], 和 Albert Rubio[1]

[1] 西班牙马德里康普顿斯大学[2] 西班牙马德里理工大学 pabgordi@ucm.es

摘要。效率是任何类型程序的一项基本属性，而对于在区块链上运行的程序（称为 *smart contracts*）而言，这一点尤为重要。这是因为，优化智能合约会直接降低合约的部署和执行成本：费用与合约的字节大小及其资源消耗（称为 *gas*）相关。优化内存使用是一项颇具挑战的问题，其中一项难点是必须准确推断被访问的内存位置。对于由使用最广泛的编译器 solc 生成的以太坊虚拟机（EVM）字节码来说，情况也是如此：其非同寻常且低层次的内存使用方式给自动推理带来了挑战。本文介绍一种在 solc 生成的 EVM 字节码层面开发的静态分析方法，可推断出不必要的内存写入访问，从而安全地将其移除。我们将该方法应用于超过 19,000 个真实智能合约，在不到 4 小时内检测出约 6,200 次不必要的写访问。有趣的是，其中许多写入涉及 solc 生成的内存使用模式；通过移除整个字节码基本块，可以大幅优化这些模式。据我们所知，现有优化工具无法推断此类不必要的写访问，因此也无法检测影响以太坊智能合约部署和执行成本的这些低效问题。

## 1 引言

以太坊虚拟机与内存模型。 以太坊 [23] 如今被认为是全球领先的可编程区块链。它提供了一种名为以太坊虚拟机（EVM）的虚拟机 [17], , 用于执行在区块链上运行的程序。这类程序称为以太坊“智能合约”，可以使用 Solidity 等高级编程语言编写 [6], Vyper [4], Serpent [3] 或 Bamboo [1]

* 本研究部分由西班牙科学与创新部、AEI 和欧洲区域发展基金（欧盟）项目 PID2021-122830OB-C41 和 PID2021-122830OA-C44 资助

然后将其编译为 EVM 字节码。EVM 字节码是最终部署到区块链上的代码，也已成为开发分析和优化工具的统一格式。先前的研究已对 EVM 程序的内存模型进行了描述 [15,16,22,23]。主要有三个可存储和访问数据的区域：(1) 以太坊虚拟机是一种基于栈的虚拟机，这意味着大多数指令都使用机器 栈中最顶端的元素进行计算。该内存区域只能容纳有限数量的值，最多为 1024 个 256 位字。(2) EVM 程序使用名为 存储 的内存区域持久化存储数据。该区域由 256 位地址到 256 位字的映射构成，其内容会在外部函数调用之间持久保存。(3) 第三个内存区域是一个局部易失性内存区，我们称之为 EVM 内存，也是本文的研究重点。该内存区域表现为一个简单的按字寻址字节数组，可按字节或以一个字为单位进行访问。EVM 内存可用于分配动态局部数据（例如数组或结构体），也可供某些特定的 EVM 字节码指令使用；这些指令要求将较长的操作数存储在局部内存中。例如，计算密码学哈希的指令，以及用于向外部函数调用传递参数和接收其返回数据的指令。编译器以不同方式使用栈和易失性内存区域。使用最广泛的 Solidity 编译器 `solc` 生成的 EVM 代码会使用栈存储值类型局部变量，以及复杂计算的中间值和跳转地址；而数组类型和用户定义的结构体类型等引用类型局部变量则存储在内存中。我们的研究重点是检测 `solc`生成的代码中不必要的内存写入访问。不过，由于该分析在 EVM 层进行，因此也很容易适配其他编译器生成的 EVM 代码。

优化。以太坊智能合约的优化是一个热门研究课题，参见例如 [8–12, 18, 20] 及其参考文献。这是因为降低其成本具有三个方面的重要意义：(1) 部署费用。当合约部署到区块链上时，所有者需支付一笔与字节码大小（以字节计）相关的费用。因此，一个明确的优化标准就是程序的字节大小。Solidity 编译器 solc [6] 将此类字节大小缩减作为优化目标。(2) 燃气计量执行。客户端在区块链上执行交易时，需要支付一笔费用。这笔费用由每笔交易的固定金额，加上执行该交易中被调用函数内所有字节码指令的成本组成。该成本以“Gas”衡量（随后按相应的加密货币计价），因此这种执行方式被称为燃气计量执行。EVM 规范（[23] 及其后续更新）为该语言中的每条字节码指令规定了精确的燃料消耗。大多数 EVM 字节码优化工具 [8–12, 18] 的目标是降低此类燃料消耗，因为这将降低智能合约上所有交易的价格。(3) 扩展以太坊的处理能力。由于当前需求处理的交易量极其庞大，人们非常关注扩展以太坊网络的处理能力，以增加其能够处理的交易数量。总体而言，对 EVM 字节码进行优化——

尤其是优化其内存使用——是朝这一方向迈进的重要一步。

挑战与贡献。优化内存使用被认为是一项颇具挑战性的问题，需要精确推断所访问的内存位置；而这通常取决于所分析语言的内存模型，以及生成待执行代码的编译器。对于由`solc` 编译器生成的以太坊智能合约，其内存模型相当特殊，底层内存使用模式给自动推理带来了挑战。一方面，内存分配并非通过一条分配指令完成，而是由一系列指令执行，这些指令使用存储在地址 $0x40$处的值作为空闲内存指针，即指向可用于分配新内存的第一个内存地址的指针。一般而言，内存被组织为一系列槽位：一个槽位由若干连续的内存位置组成，在字节码中通过同一个起始内存位置及相应的偏移量访问。一个槽位可能仅用于存放智能合约中创建的数据结构；但在使用嵌套数据结构时，我们也可以从一个槽位找到指向其他内存槽位的指针，这些槽位存放着嵌套组件。最后，还有其他类型的瞬态槽位，用于存放临时数据，也需要通过精确的内存分析加以捕获。这些特性构成了推断不必要写访问的主要挑战。为准确处理这些问题，我们作出以下主要贡献：(1) 提出一种槽位分析，用于（过度）近似执行过程中创建的槽位及其分配所在的程序点；(2) 随后引入一种槽位使用分析，根据字节码指令推断对不同槽位的访问；(3) 最后推断不必要的写访问，即程序中写入内存、但之后没有任何指令读取该内存的位置；(4) 实现该方法，并在真实智能合约上进行全面的实验评估，检测出由`solc`生成的、属于高度可优化内存使用模式的不必要写访问。最后值得一提的是，内存分析（第 1 和第 2 点）的应用不止于检测不必要的写访问：精确的 EVM 内存模型对于提升任何后续分析的准确性都至关重要（例如，参见 [16] ，了解内存分析的其他具体应用）。

## 2 内存布局与示例引入

内存操作码。以太坊虚拟机的指令集包含常见的内存访问指令：最基本的内存操作指令是 `MLOAD`和 `MSTORE`，分别用于从内存中加载和存储一个 32 字节字。[3] solc 编译器生成的代码采用累积模型处理内存：内存会随着程序的执行而分配，且永不释放。与 Java 虚拟机等其他字节码虚拟机不同，

[3] 虽然使用 MSTORE8 指令时，局部内存是按字节寻址的，但为简化描述，我们只考虑通常的按字寻址 MSTORE 情况。

```
1 结构体 TokenOwnership {
2     地址 addr;
3     无符号64位整数 startTs;
4     布尔型 burned;
5 }
6
7 合约 运行中1 {
8   //
9   函数 unpackedOwnership
10      ( uint256 packed ) 公开
11 S1S2   返回 (TokenOwnership
        内存 所有权) {
12      所有权 . 地址 =...;
13      所有权 . 起始时间戳 =...;
14      所有权 . 已销毁 = ...;
15    }
16 }
```

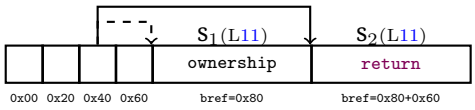
```
17 合约 Running2 {
18   Running1 c ;
19   映射(uint256=>uint256 ) 私有 _打包所有权 ;
20   // ..
21 秒6 函数 _ownershipAt(uint256 i ) private
22 秒7     返回 (TokenOwnership 内存) {
23     返回 c. unpackedOwnership( 打包所有权 [i ]);
24   }
25
26 秒3 函数 explicitOwnershipOf(uint256 tokenId )
27 秒4       公开返回值(TokenOwnership 内存) {
28 秒5   TokenOwnership 内存 所有权;
29 秒8   if (...) { return 所有权; }
30   所有权 = _ownershipAt( tokenId );
31 秒5   // ...
32   return 所有权;
33 }
```

图 1：智能合约 ERC721A 的摘录。

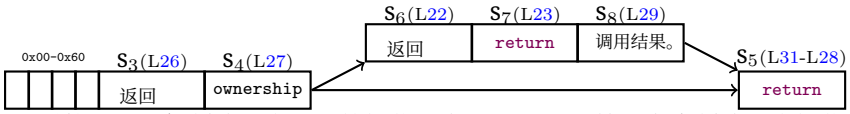
以太坊虚拟机没有专门用于分配内存的指令。内存分配通过一系列指令完成，这些指令使用存储在地址 $0x40$ 处的值作为空闲内存指针，即指向可用于分配新内存的第一个内存地址的指针。下文中，我们用$mem\langle x \rangle$ 表示存储在内存位置 x 处的内容。

内存槽。在一般情况下，内存由一系列槽位组成。一个槽位由连续的内存位置构成，可通过其起始内存位置访问。我们将其称为该槽位的基准引用（简称 baseref），再加上访问槽位中特定位置所需的相应偏移量。槽位通常用于存储 Solidity 程序中创建的某种数据结构（例如数组或结构体）的全部或部分内容，且其长度是已知的。

示例 1（存储槽）。图 1 展示了智能合约 ERC721A [2]的一个摘录，其中包含两个不同的合约 `Running1` 和 `Running2`。我们省略了不相关的指令，例如第 12-14 行（简称 L12-L14）中的指令。图1 左侧的合约`Running1` 包含公共函数 `unpackedOwnership`，该函数返回一个在 L1-L5定义的 TokenOwnership 类型结构体。右侧所示的合约`Running2`包含公共函数`explicitOwnershipOf`；根据某个不相关的条件，该函数返回一个空的 TokenOwnership 类型结构体（L28），或返回从对合约`Running1` 的函数`unpackedOwnership` 的调用中获得的 `TokenOwnership`（L23）。该调用在私有函数 `_ownershipAt`中执行。合约 `Running1` 中函数`unpackedOwnership`的执行在 L11处分配了两个不同的内存槽：$S_1$, 用于返回变量`ownership`，$S_2$, 则用于实际从函数返回`ownership`的内容：




Running2 中的函数 `explicitOwnershipOf` 对内存的使用更加密集，可从以下图示中看出：`Running2`




执行此函数最多可能创建六个不同的槽位。在 L26 和 L27处，它会创建两个槽位：一个用于函数头的 返回 部分中声明的结构体（$s_3$），另一个用于局部变量 所有权（$s_4$）。根据`if` 语句中条件的求值结果，它可能会创建执行对 `_ownershipAt` 的调用所需的槽位，进而创建对 `Running1.unpackedOwnership`的外部调用所需的槽位。对私有函数的调用涉及三个槽位：一个用于在 L29处的 `_ownershipAt` 的 返回 部分中声明的结构体（$s_6$），一个用于在 L23 处管理外部调用数据的槽位（$s_7$），以及一个用于在 L29处存储私有函数 `_ownershipAt` 结果的槽位（$s_8$）。最后，会创建一个新槽位（$s_5$），用于返回`explicitOwnershipOf`的结果。根据`if` 条件的求值结果，这个新槽位可能包含 $s_4$ 或 $s_8$, 的内容。

分配 $t$ 内存时，会通过读取并递增空闲内存指针 ($mem\langle 0x40\rangle$) $t$ 位置来预留槽位。从这次更新开始，使用刚分配槽位的基准引用，后续对该槽位的访问则通过此基准引用进行，也可以在其上增加偏移量。

示例 2（内存槽预留）。以下 EVM 代码片段为 `TokenOwnership` 类型分配了一个槽位。EVM 字节码执行三个步骤：
(i) 加载空闲内存指针的当前值 $mem\langle 0x40\rangle$，该指针将用作新槽位的基准引用；(ii) 将 $t$ 加到基准引用上，计算新的空闲内存地址；(iii) 将新的空闲内存指针存储到 $mem\langle 0x40\rangle$ 中。此外，在控制流图的同一基本块中，槽位预留之后紧接着在 `0x19A`、`0x1AB` 和 `0x1B4` 处进行槽位初始化。

```
0x175: JUMPDEST
0x176: PUSH1 0x40
0x178: MLOAD     // (i) baseref
       DUP1
       PUSH1 0x60 // Sizeof "t"
       ADD   // (ii) baseref+0x60
0x17D: PUSH1 0x40
0x17F: MSTORE    // (iii)
       ...
0x19A: MSTORE    // baseref+0x00
       ...
0x1AB: MSTORE    // baseref+0x20
       ...
0x1B4: MSTORE    // baseref+0x40
```

Solidity 引用类型值（例如数组、结构体类型变量和字符串）使用这种通用模式存储在内存中，但存在一些细微差异。不过，在某些情况下，上述步骤会有所变化，存储槽的大小也无法预先确定，因此此时无法更新空闲内存指针。例如，当外部调用返回数据时，其长度事先未知，因此只有在写入指针所指向的内存后，才会更新空闲内存指针。在其他情况下，空闲内存会被用作生命周期较短的临时区域，例如向外部调用传递参数时，此时不会更新空闲内存指针。必须通过精确的内存分析来检测这些通用模式的变体。为此，我们认为，当某个存储槽的基准引用已从 $mem\langle 0x40\rangle$ 中读取，但空闲内存指针尚未更新时，该存储槽处于瞬态状态；而当空闲内存指针已向前推进时，该存储槽处于永久状态。

示例 3（瞬态槽）。现在我们聚焦于 Running2 的 L23 中的外部调用，该调用执行一次 STATICCALL，并从栈中读取（详见 [23] ）输入参数的内存位置以及保存调用结果的位置。有趣的是，这两个位置复用了同一个存储槽（对应于 $s_7$），如下列来自 _ownershipAt 的 EVM 字节码所示：

```
    PUSH4 0xb04dd20b // func. selector
    ...
    PUSH1 0x40
0x114: MLOAD   // baseref transient slot
    ...
    DUP2
    MSTORE  // stores func. selector
    PUSH1 0x04
    ADD     // offset of funct. args.
    ...   // copy func. args.
    MSTORE  // stores func. args.
    ...
```

```
    PUSH 0x40
0x132: MLOAD      // slot baseref
    ...
0x139: STATICCALL // external call
    ...
    PUSH1 0x40
0x151: MLOAD     // slot baseref
    RETURNDATASIZE
    ...
    ADD       // baseref + data size
    ...
0x15E: PUSH1 0x40
0x160: MSTORE   // permanent slot
```

调用首先读取空闲内存指针（位于 0x114），并将参数数据存储到该地址（其中第一个参数是函数选择器）。重要的是，写入输入参数时不会向前移动指针，因此该槽仍处于瞬态状态。执行 0x139 处的调用后，结果会从基准引用开始写入内存（覆盖输入参数使用的位置），随后再次读取空闲内存指针（0x151）并将其更新（0x160），加上实际返回数据大小（RETURNDATASIZE），从而最终使该槽永久生效。

将数据从公共函数返回给外部调用者时，也会使用临时槽。在这种情况下，公共函数的以太坊虚拟机代码会使用 RETURN 指令停止执行。它会从栈中读取存放待返回数据长度和数据的内存位置。然而，它不会更改 $mem\langle 0x40\rangle$，因为函数代码会在此处停止执行，这一点可以从 explicitOwnershipOf 的以太坊虚拟机代码中看出（对应槽位 $s_5$）：

```
    PUSH1 0x40
0x4D: MLOAD    //ret slot baseref
    ...
    MSTORE  // ret.addr (ret+0x00)
    ...
    MSTORE  // ret.startTs (ret+0x20)
    ...
    MSTORE  // ret.burned (ret+0x40)
```

```
    ...
    PUSH1 0x40
0x5A: MLOAD  //ret slot revisit
    DUP1
    SWAP2  //Baseref of ret plus size
    SUB    //Size of ret data
0x5E: SWAP1
0x5F: RETURN //ret returned
```

返回槽的基准引用被读取（位于0x4D），并作为临时槽使用：通过为结构体中包含的每个字段添加相应偏移量，将要返回的结构体内容写入其中（左列中的指令）。左侧代码以基准引用和已存储数据大小位于栈顶结束。之后再次读取基准引用（右列顶部），并在调用RETURN 指令之前计算返回数据的长度（用基准引用加上已存储数据的大小所得值减去基准引用）。

## 3 不必要的写访问推断

本节介绍我们对不必要的写访问进行的静态推断。我们首先在第3.1节中介绍有关控制流图（CFG）类型以及

所依赖的静态分析。随后，分析分为连续的三个步骤：（1）槽位分析，在第3.2节中介绍，用于识别执行过程中创建的槽位以及分配这些槽位的程序点；（2）槽位使用分析，在第3.3节中介绍，用于计算上一步所识别的不同槽位上的读取和写入访问；以及（3）不必要写入访问检测，在第3.4节中介绍，用于找出这样的程序点：该处对某个槽位进行了写访问，而之后不再对其进行读取访问。

### 3.1 上下文敏感控制流图与流敏感统计分析

以太坊智能合约控制流图的构建是任何反编译器和静态分析工具的关键组成部分，也是此前研究的主题 [13,14,21]。控制流图越精确，我们的分析结果就越准确。尤其是，控制流图构建中的上下文敏感性 [14] 对于获得精确结果至关重要。我们通过克隆从不同上下文到达的基本块来实现上下文敏感性。

示例 4（上下文敏感控制流图）。`Running2` 的 EVM 代码为处理 `TokenOwnership` 类型的结构体创建了多个槽位。有趣的是，所有这些槽位都是通过示例 2 中所示的同一段 EVM 代码创建的，该代码对应于从程序点 `0x175` 开始的控制流图基本块。由于该基本块是从不同上下文到达的，上下文敏感控制流图包含此基本块的三个克隆：`0x175`，其在 L26 处创建 $s_3$；`0x175_0`，其在 L27 处创建 $s_4$；以及`0x175_1`，其在 L22 处预留 $s_6$, 。基本块克隆意味着程序点也会被克隆，因此我们采用相同的下标表示法来指代克隆基本块中包含的程序点：例如，程序点 `0x178`包含`MLOAD 0x40`，用于获取在基本块 `0x178`处预留槽位的基准引用；而`0x178_0` 则指向位于`0x178_0`处的同一个`MLOAD`，依此类推。

下文中，我们假设克隆已经完成，因此，使用所得控制流图（包含克隆）的内存分析也具有上下文敏感性，无需额外扩展。与标准分析一样 [19], ，必须定义抽象状态的概念，用以描述分析过程中收集的抽象信息；还必须定义传递函数，用以模拟每种可能输入对应的分析输出。除了上下文敏感性之外，我们将在接下来的两节中介绍的两种分析也都是流敏感的，也就是说，它们会对程序的控制流图进行流敏感遍历，并在分析控制流图中的每个基本块时，将其调用者推断出的信息作为输入。当分析在控制流图的某个基本块处遇到新信息时，我们使用操作 $\sqcup$ 合并两个抽象状态，并使用运算符 $\sqsubseteq$ 检测是否已达到不动点，从而判断分析是否终止。操作 $\sqcup$ 和 $\sqsubseteq$、抽象状态以及传递函数，将针对每种具体分析分别定义。

### 3.2 槽位分析

槽位分析旨在推断抽象槽位，它们是对程序执行过程中将进行的所有内存分配的抽象。

推断出的槽位是抽象的，因为在分配槽位的程序点层面进行了过近似。因此，一个抽象槽位可能代表多个（不一定连续的）实际内存槽，例如，在循环内分配内存时。槽位分析将查找读取 $mem\langle 0x40\rangle$ 中所存值以预留内存空间的程序点。分析这些程序点有两个原因：首先，用于获取内存槽的基准引用；其次，从这一程序点起，相应槽位的内存预留便已开始，并将在后续某个程序点变为永久预留。槽位分析的输出是一个包含已分配抽象槽位的集合，在下文定义 2 中记作 $\mathcal{S}_{all}$ 。每个已分配的抽象槽位（即2 个定义中的每个元素）都是一个程序点集合，因为同一个抽象槽位可能对应多个程序点，在这些程序点上， $mem\langle 0x40\rangle$ 会在其预留变为永久之前被读取。为了获得 $\mathcal{S}_{all}$，内存分析会对程序的（上下文敏感）控制流图进行流敏感遍历，在每个程序点维护临时槽位集合（即其基准引用已被读取，但尚未变为永久的槽位），并将定义 1 中的传递函数应用于基本块中的每条字节码指令，直至达到不动点。分析的一个抽象状态是集合 $\mathcal{S} \subseteq \wp(\mathcal{P}_R)$，其中 $\mathcal{P}_R$ 是所有读取 $mem\langle 0x40\rangle$ 的程序点的集合。程序分析从所有程序点上的 $\mathcal{S} = \{\emptyset\}$ 开始，并将 $\sqcup$ 和 $\sqsubseteq$ 作为集合并与包含操作。由于程序点数量有限，且 $\wp(\mathcal{P}_R)$也有限，因此显然可以保证终止。下文中，Ins 是 EVM 指令的集合。为简单起见，我们将`MLOAD 0x40` 和`MSTORE 0x40` 视为 Ins 中的单条指令。

定义 1（槽位分析传递函数）。给定程序点 $pp$ 以及一条指令 $I \in$ Ins、一个抽象状态 $\mathcal{S}$ 和 $\mathcal{K} = \{$*MSTORE 0x40*, *RETURN*, *REVERT*,*STOP*, *SELFDESTRUCT*$\}$，槽位分析传递函数 $\nu$ 定义为映射 $\nu : \text{Ins} \times \wp(\mathcal{S}) \mapsto \wp(\mathcal{S})$，其计算依据如下表：

| | I | $\nu$(I $\mathcal{S}$) |
|---|---|---|
| (1) | *MLOAD 0x40* | $\{s \cup \{pp\} \mid s \in \mathcal{S}\}$ |
| (2) | $I \in \mathcal{K}$ | $\{\emptyset\}$ |
| (3) | *otherwise* | $\mathcal{S}$ |

下面直观地解释上述传递函数的工作原理。如第 2 节所述，在 EVM 程序中，所有内存预留都始于读取$mem\langle 0x40\rangle$ ，该读取通过一条 `MLOAD` 指令完成，且其前面有一条 `PUSH 0x40` 指令（定义 1 中的情况 1）。在这种情况下，传递函数会将当前程序点的所有集合都加入 $\mathcal{S}$ ，因为原则上，这会访问当前程序点上已开放但尚未永久化的相同存储槽。为了正确识别这些存储槽，我们的分析还会查找那些将槽位预留永久化的程序点（定义 1 中的情况 2），也就是包含指令 $I \in \mathcal{K}$ 的程序点。最常用于将槽位预留永久化的指令，是使用 `MSTORE` 对 $mem\langle 0x40\rangle$ 进行写访问；该指令会推进空闲内存指针，使得后续任何对$mem\langle 0x40\rangle$ 的读取访问都会分配不同的存储槽。 $\mathcal{K}$ 中的其余指令则以不同方式结束执行（正常返回、强制停止、回滚执行等）。在所有这些情况下，都需要将该存储槽视为永久存储槽，以便

之后我们可以推理其中涉及的潜在不必要写访问。这些指令执行后，集合 $\mathcal{S}$ 为空，因为所有临时（抽象）存储槽都已永久化。我们用记号 $\mathcal{S}_{pp}$ 表示在程序点 $pp$计算出的抽象状态。

示例 5（槽位分析）。Running2 的槽位分析在所有程序点均从 $\mathcal{S}_{pp}$={∅} 开始。当分析到从 0x175 开始的基本块时（见示例 2），$\mathcal{S}_{0x175}$ 为 {∅}，并且一直为空，直到 0x178；此处读取了 $s_3$ 的基准引用，因此 $\mathcal{S}_{0x178}$={{0x178}}。在 0x17F 更新空闲内存指针后，该槽位变为永久槽位，因此有 $\mathcal{S}_{0x17D}$={{0x178}} 和 $\mathcal{S}_{0x17F}$={∅}。遵循相同模式，$s_4$ 和 $s_6$ 分别在指令 0x178_0 和 0x178_1 处被保留，并在 0x17F_0 和 0x17F_1 处关闭（位于克隆的基本块中）。另一方面，$s_5$ 的基准引用在两个连续的程序点（0x4D 和 0x5A）处被读取，并在 0x5F 处更新，因此我们有 $\mathcal{S}_{0x4D}$={{0x4D}}，且这种情况持续到 $\mathcal{S}_{0x5A}$={{0x4D,0x5A}}，之后又保持相同状态，直到 $\mathcal{S}_{0x5F}$={∅}。最后，执行 STATICCALL 后（见示例 3），我们在 0x114、0x132 和 0x151 处连续三次读取 $mem\langle 0x40\rangle$，它们指向同一存储槽 $s_7$，该槽在 0x160 处变为永久槽位。因此，我们有 $\mathcal{S}_{0x151}$={{0x114, 0x132, 0x151}} 和 $\mathcal{S}_{0x160}$ = {∅}。

如第 3.1 节所述，利用传递函数，我们对程序的（上下文敏感）控制流图进行流敏感遍历，并使用推断出的调用者信息作为输入来分析每个基本块。达到不动点后，我们便获得了每个程序点的抽象状态，并据此计算程序中分配的抽象槽位集合，记为 $\mathcal{S}_{all}$。

定义 2。已分配抽象槽位集合 $\mathcal{S}_{all}$ 定义为$\mathcal{S}_{all} = \bigcup_{pp\in\mathcal{P}_W} \mathcal{S}_{pp-1}$，其中 $\mathcal{P}_W$ 是所有满足 $I\in\mathcal{K}$的程序点 $pp$:$I$ 的集合。

示例 6（$\mathcal{S}_{all}$ 计算）。根据例 5 中的 $\mathcal{S}_{0x17F-1}$、$\mathcal{S}_{0x17F_0-1}$、$\mathcal{S}_{0x17F_1-1}$、$\mathcal{S}_{0x160-1}$和 $\mathcal{S}_{0x5F-1}$ 的值，对 Running2进行槽位分析后，我们得到：$\mathcal{S}_{all}$={{0x178}, $\underbrace{\{0x178_0\}}_{s_3}$, $\underbrace{\{0x178_1\}}_{s_4}$, $\underbrace{\{0x114, 0x132, 0x151\}}_{s_6}$, $\underbrace{\{0x5A, 0x4D\}}_{s_7}$, $\underbrace{\ldots\}}_{s_5}$.
请注意，克隆基本块0x175 使我们的分析能够在原始 EVM 代码中，为同一程序点0x178检测到三个不同的存储槽 $s_3$, $s_4$ 和 $s_6$, 。

下一个示例展示了当程序包含循环且需要通过抽象来近似存储槽时，分析的行为。

示例 7（循环）。图2 展示了合约Running3，其中包含智能合约 [2] 中的函数 explicitOwnershipsOf （通过STATICCALL调用）。该函数接收一个代币标识符数组作为参数，并返回一个TokenOwnership 结构体数组；该数组通过在循环中调用Running2 中的函数explicitOwnershipOf 填充。分析为图2 所示的合约 Running3 识别出的槽位为：$s_9$, ，用于将参数tokenIds 复制到内存；$s_{10}$, ，用于创建本地数组ownerships （L44），该数组包含数组长度以及指向结构体的指针，这些结构体最初由 $s_{11}$ 识别（之后由 $s_{13}$ 识别）；$s_{12}$，用于STATICCALL 的输入

```
37 合约        运行3 {
38   Running2 c;
39
40 秒9 //函数 explicitOwnershipsOf(uint256[]                内存 tokenIds)
41       public view returns (TokenOwnership [] 内存) {
42       未检查 {
43         uint256 tokenIds 长度       = tokenIds.length;
44 S10S11  TokenOwnership [] 内存 所有权 = new                TokenOwnership [](代币 ID 长度);
45         对于(uint256 i;      i != 代币 ID 长度; ++i) {
46 S12S13              所有权[i] = c.explicitOwnershipOf(tokenIds[i]);
47         }
48 秒14     返回      所有权;
49       }
50   }
51 }
```

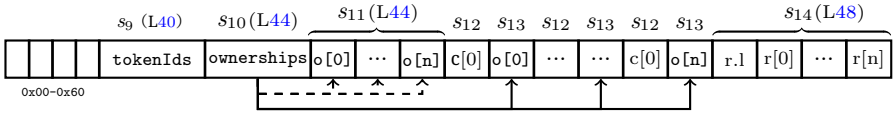



图 2：合约 `Caller` 的 Solidity 代码。

参数和返回数据（L46）；$s_{13}$，用于抽象表示存储`STATICCALL` 输出结果的结构体（L46）；以及 $s_{14}$,，其中包含所有权的长度和 $s_{13}$ 的副本，用于返回结果（L48）。重要的是，L44 处的本地数组声明会生成一个循环，分配与数组中元素数量相同的结构体。因此，$s_{11}$ 是一个抽象槽位，表示最初添加到数组中的所有`TokenOwnership`。同样，$s_{12}$ 和$s_{13}$ 是在`for` 循环内创建的，每个抽象槽位代表循环执行的每次迭代所对应的多个具体槽位。请注意，循环的每次迭代都会创建一个 $s_{12}$ 实例，以获取调用结果；随后该实例会被复制到 $s_{13}$，并由 `ownerships` （$s_{10}$）指向。

作为记号，我们将使用唯一的数字标识符（1、2、.....）来指代每个抽象槽位（在 $\mathcal{S}_{all}$ 中表示为一个集合），并通过函数 $get_id(a)$ $a \in \mathcal{S}_{all}$ 检索它。我们使用 $\mathcal{A}$ 指代程序中所有此类标识符的集合。此外，给定一个程序点 $pp$，其指令为 `MLOAD 0x40`，我们定义函数 get slots($pp$)，以检索 $\mathcal{S}_{all}$ 中可能在 $pp$ 处被引用的元素的标识符，具体如下：$get_slots(pp) = \{id \mid a \in \mathcal{S}_{all} \wedge pp \in a \wedge id = get_id(a)\}$.分析的可靠性将在下一个定理中阐述。

定理 1（槽位分析的可靠性）。集合 $\mathcal{S}_{all}$ 是程序执行期间分配的具体槽位的可靠超集。

证明基本上就是要表明，对同一存储槽的所有具体访问都由同一个抽象槽位近似（详情见附录A）。

### 3.3 槽位访问分析

尽管第 3.2 节考察了分配，分析的下一步是推断可能访问所推断抽象槽位的程序点。为此，

我们的槽访问分析需要传播保存在执行栈不同位置的抽象槽位引用。重要的是，我们跟踪的不仅是栈位置；为了抽象存储在内存中的复杂数据结构（例如结构体数组），还需要跟踪可能保存在内存位置中的抽象槽位。正如例 7所示，一个槽位中的内存位置可能包含指向另一槽位的另一个内存位置的指针，这种情况会在使用嵌套数据结构时发生。因此，抽象状态是一种映射，用于存储保存在栈位置或其他槽位内部内存位置中的潜在槽位。

定义 3（内存分析抽象状态）。内存分析抽象状态是一个形如 $\pi\ \ \mathcal{T} \cup \mathcal{A} \mapsto \wp(\mathcal{A})$ 的映射。

$\mathcal{T}$ 是包含所有栈位置的集合，我们用从 0（栈底）开始的自然数表示这些位置，而 $\mathcal{A}$ 是第3.2节计算得到的抽象槽标识符集合。我们将所有内存分析抽象状态组成的集合称为 $AS$。请注意，对于每个条目，我们会为每个栈位置保留一个潜在槽位集合，因为一个基本块可能从多个具有不同执行栈的基本块到达，例如在循环或*if-then-else* 结构中。在下文中，我们假设，对于给定的值 $k$，映射 $\pi$ 在$k \notin dom(\pi)$时返回空集。该推断由流敏感分析（如第3.1节所述）执行，该分析通过以下传递函数跟踪任意程序点所使用的抽象槽信息。

定义 4（内存分析传递函数）。给定程序点 $pp$ 处具有输入操作数 $n$ 的指令 $I$和抽象状态 $\pi$，内存分析传递函数 $\tau$定义为映射 $\tau:\ \ Ins\times AS \mapsto AS$，形式如下：

| | $I$ | $\tau(I,\pi)$ |
|---|---|---|
| (1) | *MLOAD 0x40* | $\pi[t \mapsto get_slots(pp)]$ |
| (2) | *MLOAD* | $\pi[t \mapsto \{m \mid s \in \pi(t) \wedge m \in \pi(s)\}]$ |
| (3) | *MSTORE* | $\pi[s \mapsto \pi(s) \cup \pi(t-1) \backslash \{t, t-1\}\ \ \forall s \in \pi(t)$ |

| | $I$ | $\tau(I,\pi)$ |
|---|---|---|
| (4) | *SWAPi* | $\pi[t \mapsto \pi(t-i)\ \ t-i \mapsto \pi(t)]$ |
| (5) | *DUPi* | $\pi[t+1 \mapsto \pi(t-i+1)]$ |
| (6) | $otherwise$ | $\pi \backslash x \qquad t-n < x \leq t$ |

$pp$ 是执行 $I$ 前栈顶的数值位置。

让我们解释上述定义。传递函数区分了两种不同类型的MLOAD：（1）访问位置 $mem\langle 0x40\rangle$，这些访问会通过 get slots($p$) 从先前的分析中获取可能被使用的存储槽的基准引用并将其返回；（2）其他MLOAD 指令，它们可能会返回内存位置中的槽位基准引用。因此，我们必须考虑两种情况：如果读取的内存位置存储的是通用值（例如数字），那么 $\pi(t) = \emptyset$；如果读取的内存位置可能存储一个抽象槽位，那么 $\pi(t)$ 就包含该内存位置可能存储的所有抽象槽位。关于（3），MSTORE 有两个操作数： $t$ 处的操作数是将由MSTORE修改的内存地址，$t-1$ 处的操作数则是要存储到该地址的值。对于 $\pi(t)$ 中的每个元素 $s$，分析都会添加 $\pi(t-1)$ 中的抽象槽位。分析还会处理（4–5）中所示的其他指令 SWAP* 和DUP* ，

| PP | 指令 | $\pi$ |
|---|---|---|
| 0x175_1 | JUMPDEST | $\{3\mapsto s_3, 4\mapsto s_4\}$ |
| 0x176_1 | PUSH1 0x40 | $\{3\mapsto s_3, 4\mapsto s_4\}$ |
| 0x178_1 | MLOAD | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6\}$ |
| 0x179_1 | DUP1 | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x17A_1 | PUSH1 0x60 | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x17C_1 | ADD | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x17D_1 | PUSH1 0x40 | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x17F_1 | MSTORE | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6\}$ |
| 0x180_1 | DUP1 | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| … | | |
| 0x198_1 | AND | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x199_1 | DUP2 | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6, 11\mapsto s_6\}$ |

| PP | 指令 | $\pi$ |
|---|---|---|
| 0x19A_1 | MSTORE | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| … | | |
| 0x1A9_1 | AND | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x1AA_1 | DUP2 | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6, 11\mapsto s_6\}$ |
| 0x1AB_1 | MSTORE | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| … | | |
| 0x1B2_1 | ISZERO | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x1B3_1 | DUP2 | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6, 11\mapsto s_6\}$ |
| 0x1B4_1 | MSTORE | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6, 9\mapsto s_6\}$ |
| 0x1B5_1 | POP | $\{3\mapsto s_3, 4\mapsto s_4, 8\mapsto s_6\}$ |
| 0x1B6_1 | SWAP1 | $\{3\mapsto s_3, 4\mapsto s_4, 7\mapsto s_6\}$ |
| 0x1B7_1 | JUMP | $\{3\mapsto s_3, 4\mapsto s_4, 7\mapsto s_6\}$ |

图 3：为结构体预留内存槽的控制流图基本块

它们会交换或复制参与操作的栈元素。最后，所有其他操作都会根据从栈中取出并写入栈中的元素数量，删除不再使用的栈元素（情况 6）。

示例 8（transfer）。现在我们关注基本块 0x175 的分析，如图 3 所示。正如我们已经解释过的，该基本块负责创建用于处理多个 TokenOwnership 类型结构体所需的内存，因此它在控制流图中被克隆。具体而言，我们关注克隆块 0x175_1。基本块分析从一个大小为 7 的栈开始，并在位置 3 和 4 包含抽象槽位 $s_3$ 和 $s_4$，，它们是在图 1 的 L25 和 L26 处创建的。在0x178_1，读取了 $mem\langle 0x40\rangle$，并通过 $get_slots$（0x178 1），同时考虑到 $top(0x1781)=8$，我们向 $\pi$ 添加一个新条目 $8 \mapsto \vec{} s_6$。在 0x179_1、0x180_1、0x1AA_1、0x1B3_1 处，传递函数会复制栈中存储的槽标识符。示例中的 MSTORE 和 POP 指令会从栈中移除一个槽标识符。

由于该分析具有流敏感性，CFG 中每个基本块的分析都以跳转到该基本块的各基本块通过传递函数计算出的抽象状态的并集 $\sqcup$ 作为输入，并持续应用内存分析传递函数，直到达到不动点。操作 $A \sqcup B$ 是通过操作 $\cup$对映射 $A$ 和 $B$ 中的所有条目进行合并所得的结果。当 B 包含不在 $dom(A)$ 中的条目时，或者存在条目 $v \in dom(A) \cap \text{dom}(B)$，使得 $A(v) \subseteq B(v)$ 时，操作 $\sqsubseteq$ 按预期定义为$A \sqsubseteq B$。同样，由于定义域是有限的，因此可以保证计算会终止。

示例 9（合并抽象状态）。图 1 中 explicitOwnershipOf 的 EVM 代码在两个 $s_5$ 返回 语句中都使用了，分别位于 L28 和 L31 （见示例 1）。该 EVM 代码只有一个返回基本块，它可从 if 语句的两条不同路径到达，而这两条路径对应不同的抽象状态：（1）对应 L28 的路径对应 $\pi=\{3 \mapsto s_8\}$，另一条路径（L31）对应 $\pi=\{3 \mapsto s_4\}$。我们的分析将这两个抽象状态合并，得到 $\pi=\{3 \mapsto \{s_4, s_8\}\}$。由于进行了这次合并，我们得出结论：来自 RETURN 行的指令28 和 L31 可能返回槽位 $s_4$ 或 $s_8$ 的内容。

达到不动点时，分析已为每个程序点 $pp$ 计算出一个抽象状态，以下用 $\pi_{pp}$ 表示。

示例 10（复杂数据结构）。对图 2 中代码的分析展示了它如何处理可能包含指向其他结构的指针的数据结构，例如所有权。表示变量 ownerships 的抽象槽位为 $s_{10}$，它在两个程序点通过 MSTORE 写入，分别为 $pp_1$ 和 $pp_2$，它们分别对应 Solidity 代码中的 L44 和 L46。到达 $pp_1$ 的输入抽象状态为 $\{2 \mapsto s_9, 6 \mapsto s_{10}, 8 \mapsto s_{10}, 9 \mapsto s_{11}, 10 \mapsto s_{10}\}$，而 MSTORE 的传递函数将抽象状态变为 $\pi_{pp_1} = \{2 \mapsto s_9, 6 \mapsto s_{10}, 8 \mapsto s_{10}, s_{10} \mapsto s_{11}\}$。此时，我们可以看到变量 ownerships 被初始化为空结构体；为表示这一点，我们的分析在 $\pi$ 中加入条目 $s_{10} \mapsto s_{11}$，如定义 4 中传递函数的 MSTORE 指令所述。对 $s_{10}$ 的第二次写入由 $pp_2$ 处的另一条 MSTORE 指令执行。$pp_2$ 的输入抽象状态为 $\{2 \mapsto s_9, 5 \mapsto s_{10}, 7 \mapsto s_{13}, 8 \mapsto s_{13}, 9 \mapsto s_{10}, s_{10} \mapsto s_{11}\}$，因此得到 $\pi_{pp_2} = \{2 \mapsto s_9, 5 \mapsto s_{10}, 7 \mapsto s_{13}, s_{10} \mapsto \{s_{11}, s_{13}\}\}$。有趣的是，在 $pp_2$ 处，我们检测到 $s_{11}$ 也可能存储由调用 c.explicitOwnershipOf(tokenIds[i]) 返回的结构体，该调用由 $s_{13}$ 标识，并将其加入 $s_{10} \mapsto \{s_{11}, s_{13}\}$。最后，在方法末尾读取 $s_{10}$，返回集合 $\{s_{11}, s_{13}\}$，以将 ownerships 的内容复制到 $s_{14}$，即 return 中使用的存储槽。

可靠性保证，同一具体槽中的内存访问会由同一个抽象槽位近似表示。这可直接由定理 3（详见附录 A）。

定理 2（槽访问分析的可靠性）。$\pi_{pp}$ 是对指令 $pp$所访问槽位的可靠过近似。

### 3.4 不必要的写入内存访问推断

根据前述分析的结果，我们可以计算映射 $\mathcal{R}$ 和 $\mathcal{W}$，其形式为 $pp \mapsto \wp(\mathcal{A})$，分别用于捕获不同程序点处可能被读取或写入的槽位。为此，由于多个以太坊虚拟机指令（例如 RETURN、CALL、LOG、CREATE、.....）可能会读取或写入内存位置，并从栈中获取具体位置，因此我们定义函数 mr($I$) 和 mw($I$)：给定一条以太坊虚拟机指令 $I$，它们分别返回 $I$将要读取和写入的地址在栈中的位置。如果该指令不读取或写入任何内存位置，则函数 $mr(I) = \perp / mw(I) = \perp$。例如，$mr$(MLOAD) $= 0$，因为它读取栈顶，而 $mw$(MLOAD) $= \perp$；或者 $mr$(STATICCALL) $= 2$，且$mw$(STATICCALL) $= 4$。现在，我们定义读写映射 $\mathcal{R}/\mathcal{W}$:

定义 5（内存读写访问映射）。给定一个 EVM 程序 P，使得 $pp \equiv I \in P$ 且为 $t{=}top(pp)$，我们定义映射 $\mathcal{R}$ 和 $\mathcal{W}$ 如下：

$$\mathcal{R}(pp) = \begin{cases} \emptyset & mr(I) = \perp \\ \pi_{pp-1}(t - mr(I)) & otherwise \end{cases} \qquad \mathcal{W}(pp) = \begin{cases} \emptyset & mw(I) = \perp \\ \pi_{pp-1}(t - mw(I)) & otherwise \end{cases}$$

示例 11（$\mathcal{R}/\mathcal{W}$ 映射）。让我们说明如何计算 $\mathcal{R}$(0x139) 和$\mathcal{W}$(0x139)，其中包含 Running2 的 STATICCALL。根据分析得到的信息，我们有 $top$(0x139) $= 16$ 且 $\pi_{\texttt{0x138}} = \{3 \mapsto s_3, 4 \mapsto s_4, 7 \mapsto s_6, 10 \mapsto s_7, 12 \mapsto s_7, 14 \mapsto s_7\}$，因此得到 $\mathcal{R}$(0x139) $= \{s_7\}$和 $\mathcal{W}$(0x139) $= \{s_7\}$，也就是用于管理外部调用输入和输出的存储槽。类似地，我们得到 $\mathcal{R}$(0x178) $= \{s_3\}$ 且 $\mathcal{W}$(0x178) $= \emptyset$。

分析的最后一步是搜索那些之后永远不会被读取的存储槽写访问。为此，我们使用在 $\mathcal{R}$ 和 $\mathcal{W}$ 中计算的信息。给定程序的控制流图以及两个程序点 $p$ 和 $p2$，我们定义函数 $reachable(p, p2)$，当控制流图中存在从 $p$ 到 $p2$ 的路径时，该函数返回 $true$ 。我们将写入泄漏集合 $\mathcal{N}$ 定义如下：

定义 6. 给定一个 EVM 程序及其 $\mathcal{W}$ 和 $\mathcal{R}$，我们将 $\mathcal{N}$ 定义为$\mathcal{N} = \{pw$: $s \mid pw \in P \wedge s \in \mathcal{W}(pw) \wedge \neg exists_read(pw, s)\}$其中 $exists_read(pw, s)$ $\equiv \exists\, pr \in dom(\mathcal{R}) \mid s \in \mathcal{R}(pr) \wedge reachable(pw, pr)$.

直观地说，集合 $\mathcal{N}$ 包含取自 $\mathcal{W}$ 的那些写访问，它们不会被控制流图中的后续基本块读取。由于函数 reachable 以及集合 $\mathcal{W}$ 和 $\mathcal{R}$ 都是过近似，对 $\mathcal{N}$ 的计算会给出可安全移除的写访问，如下一个示例所示。

示例 12. 我们的分析检测到，在程序点 0x19A、0x1AB 和 0x1B4处存在 MSTORE 操作，这些操作在控制流图的后续基本块中从未被读取。这些操作对应于 $s_3$, 的内存初始化，该初始化在图 1 的代码 L26 处执行（参见例 2）。鉴于这些写访问是该存储槽的唯一用途，因此可以安全地移除整个预留。此外，分析检测到，程序点 0x19A_1、0x1AB_1 和0x1B4_1（对应于在 L22处对 $s_6$ 进行的预留）被判定为不必要。本质上，这意味着 $s_3$ 和 $s_6$ 虽已分配并初始化，却从未在程序中使用。请注意，所有这些程序点都属于两个克隆的基本块：(0x175 和 0x175_1)。然而，同一基本块的另一个克隆（0x175_0）中的三个 MSTORE 操作对应于 L27 处的分配，并未被判定为未读取，因为它们可能会在函数返回时使用。要将这些 MSTORE 操作识别为不必要，需要具备精确的上下文敏感控制流图。因此，我们无法删除该基本块，因为其中一个克隆仍然需要它；但仍可通过删除另外两种情况下跳转到该基本块的无条件跳转，对 EVM 代码进行重要优化，从而完全避免执行所有这些指令及其相应的燃料消耗 [23]。

作为前几节分析可靠性的推论，我们得出推断的写访问是不必要的。

推论1（冗余写访问推断的可靠性）。如果 $pp \in \mathcal{N}$，那么不存在这样的程序执行：在执行 $pp$ 处的写指令后，会读取在 $pp$ 处访问的存储槽。

## 4 实验评估

本节报告了我们的方法的实验评估结果，方法详见第3节。分析的所有组件均使用 Python 实现，且为开源，可从 GitHub 下载；其中提供了详细的安装和使用说明[4]。我们使用外部组件构建控制流图（因为这并非本研究的贡献）。我们的分析工具接受以 Solidity 0.8.17 及更早版本编写的智能合约，以及以太坊虚拟机 v1.10.25[5] 的字节码。实验在配备 AMD Ryzen Threadripper PRO 3995WX 64 核处理器和 512GB 内存、运行 Debian 5.10.70 的机器上进行。为了对分析进行实验评估，我们从 etherscan.io [5] 获取了以太坊合约，这些合约对应于截至 2022年7月14日源代码可用的、最近 5,000 个经验证的开源地址。其中，2.18% 地址对应的代码在使用 solc 编译时会报错。对于其余 4,891 个地址对应的代码，控制流图生成过程（这并非本研究的贡献）在其中 626 个地址上超过 120 秒后仍未完成。剔除这些失败案例后，我们最终分析了 19,199 个智能合约，因为每个地址和每个 Solidity 文件中都可能包含多个合约。请注意，84.86% 的合约使用 solc 0.8 版本编译，推测采用了最先进的编译技术。所用的完整数据集可在上述 GitHub 链接中找到。

为了让情况对我们最不利，我们在执行 `solc`优化器后运行内存分析，也就是说，我们分析的字节码可能已经经过 `solc`自带的优化器优化。这也让我们能够观察采用我们的方法是否可以进一步优化。遗憾的是，我们无法在运行超级优化器 GASOL [8], 后应用我们的工具，因为它不会生成优化后的字节码，而只会报告每个基本块在 Gas 和/或代码大小方面的收益。尽管如此，第 5 节详细比较了 GASOL 所采用的技术与我们的方法，并说明 GASOL 无法发现我们识别出的任何不必要访问。在分析的 19,199 个合约中，分析推断出 679,517 个抽象内存槽位，并在 12,803 秒内检测到 6,242 次不必要的内存写入访问。这些不必要的访问出现在绑定到 780 个不同地址的代码中，即占已分析地址的 15.95%。下表列出了包含不必要内存访问的最常见公共函数。

| 函数 | #Acc | #C | 函数 | #Acc | #C |
|---|---|---|---|---|---|
| transfer | 1745 | 441 | 释放 | 68 | 24 |
| transferFrom | 1736 | 439 | poke | 60 | 1 |
| 安全转账自 | 162 | 52 | withdraw | 54 | 32 |
| reflectionFromToken | 105 | 6 | 领取分红 | 54 | 10 |
| ERC-1155 批量接收回调 | 97 | 13 | ERC-1155 接收回调 | 48 | 12 |

“#Acc”列显示在每个函数中识别出的不必要访问次数，“#C”列则显示包含这些函数的不同合约数量。

[4] https://github.com/costa-group/EthIR/tree/memory_optimizer/ethir [5] 截至 2022 年 10 月发布的最新版本。

其中一些函数，例如 transferFrom、transfer、reflectionFromToken 或 withdraw，广泛用于基于 ERC 代币的合约实现。对这 10 个公共函数进行人工检查后，我们发现了不必要访问的两个不同来源：其中一些源于低效的编程实践，另一些则由编译器生成，且可以改进。就编译器的低效之处而言，我们检测到有些字节码会分配无法访问、也无法使用的内存槽，因为用于访问这些槽位的基准引用没有保存在栈中。例如，函数返回结构体时，总会为该数据分配内存。然而，如果函数头中未命名返回变量，编译器仍会为该数据分配内存，尽管它永远不会被访问。如果程序员了解这一行为，就能避免生成这类无用内存；更理想的是，可以在编译器中改进这种内存使用模式。例如，图 1 中的 L22 和 L26就体现了这一点，其中的函数没有为返回变量命名。因此，编译器为这些 匿名数据结构分配了永远不会使用的内存。类似地，在涉及外部调用的多种情形下，编译器也会创建永远不会使用的内存。当外部调用不返回任何结果时，编译器会创建两个内存槽：一个用于接收调用结果，另一个用于将可能的结果复制到一个永远不会使用的内存变量中。最后，对于用于货币转账的低级普通调用，编译器也会创建永远不会使用的内存。即使合约代码不会使用低级调用返回的第二个结果，编译器仍会生成用于获取该结果的代码。我们通过推断不必要的写访问发现了所有这些潜在的优化机会，并将把它们告知 `solc`开发者。

## 5 结论与相关工作

我们提出了一种针对以太坊智能合约的新型内存分析方法，并将其用于推断不必要的内存写入访问。我们在超过 19,000 个真实智能合约上应用该实现，检测到了一些会引入不必要写访问的编译模式；只需在编译器中轻松修改这些模式，就能生成更高效的代码。下面从两个方面讨论相关工作：（1）内存分析；（2）内存优化。关于（1），[14] 中对以太坊虚拟机内存的静态建模与第 3.2 节和第 3.3 节所述的内存分析有些相似之处，因为两者都旨在对内存进行建模，只是应用目的不同。另一方面，两者所采用的静态分析类型有所不同：[14] 基于 Datalog 分析，而我们定义了一个标准传递函数，并在流敏感分析中使用该函数。更重要的是，两种分析的精度有所不同。我们能够准确地对嵌套数据结构分配的内存建模，其中内存包含指向其他内存槽的指针，而 [14] 无法捕获此类访问。要进行内存优化，这一点至关重要：正如本文运行示例所示，它能够检测出不必要的写访问

，否则这些访问就会被遗漏。最后， [14], 并未研究将内存分析应用于优化，而这正是我们工作的重点。

关于（2），优化内存使用是一项颇具挑战性的研究问题，需要精确推断正在访问的内存位置。有时，这些位置在静态阶段即可确定（例如，访问 EVM 空闲内存指针时）；但正如我们所见，通常需要进行精确而复杂的推断，才能确定每条内存访问字节码所访问的存储槽。超级优化器 GASOL 的近期工作[8] 能够在控制流图（CFG）的每个基本块层面（即块内）执行一些内存优化。我们的工作与 GASOL 之间有三个根本区别：首先，只有当被寻址的内存位置指向相同的常量方向时，GASOL 才能应用这些优化。换言之，它并未真正进行内存分析（具体见第3.2 节和3.3节）。其次，这些优化仅在块内层面应用，因此错失了许多优化机会。这两点与我们的方法有本质区别，因为检测到的可优化模式（见第4节）需要跨块分析和精确的槽访问分析，因此 GASOL 无法检测到这些模式。

最后，正如第 1 节所述，除了动态内存，智能合约还会使用一种称为存储的持久化内存。关于将我们的方法应用于推断存储中的冗余访问，有两点需要说明。首先，无需开发静态分析来检测存储中的槽位，因为这些槽位在编译时已知（因此无需进行第 3.2 和 3.3 节中的推断），也就是说，可以轻松确定定义 6 中的读写集合。因此，可以轻松地为存储定义我们分析所需的读写集合。第二，由于存储是持久化内存，即使智能合约中不再有后续读取，存储写访问也不能移除，因为该数据需要保留，以供未来的交易使用。可移除的存储写访问仅限于两次写访问之间被再次写入且未被读取的访问。将这一点纳入我们的实现十分简单。不过，这种情况相当少见，我们认为实际发现的案例寥寥无几，因此能实现的优化也很有限。

## 参考文献

1. Bamboo。https://github.com/pirapira/bamboo。2. ERC721A。https://etherscan.io/address/0xfcd5c0ef90715dc052dad6de08efda758aa09f60#code。3. Serpent。https://github.com/ethereum/wiki/wiki/Serpent。4. Vyper。https://github.com/ethereum/vyper。5. Etherscan。https://etherscan.io，2018。6. Solidity 文档，2021。https://docs.soliditylang.org/en/latest/index.html。7. Elvira Albert、Jes´us Correas、Pablo Gordillo、Alejandro Hern´andez-Cerezo、Guillermo Rom´an-D´ıez 和 Albert Rubio。智能合约分析：从以太坊虚拟机到可靠的控制流图。技术报告，2020。8. Elvira Albert、Pablo Gordillo、Alejandro Hern´andez-Cerezo 和 Albert Rubio。用于处理内存与存储的 EVM Max-SMT 超级优化器。载于 Dana Fisman 和 Grigore Rosu 编，*28th International Conference on Tools and*

*Algorithms for the Construction and Analysis of Systems TACAS 2022. Proceedings* 13243 *Lecture Notes in Computer Science* 201 219 码–。施普林格，2022。9. T. Brandst¨atter、S. Schulte、J. Cito 和 M. Borkowski。Solidity 智能合约中的效率优化特征。在 *2020 IEEE International Conference on Blockchain (Blockchain)* 281 290, 2020 ，页码–。10. Ting Chen、Youzheng Feng、Zihao Li、Hao Zhou、Xiapu Luo、Xiaoqi Li、Xiuzhuo Xiao、Jiachi Chen 和 Xiaosong Zhang。Gaschecker：用于发现 Gas效率低下的智能合约的可扩展分析。*IEEE Transactions on Emerging Topics in Computing* 99 1 14, 03 2020 ，PP( )：–。11. Ting Chen、Zihao Li、Hao Zhou、Jiachi Chen、Xiapu Luo、Xiaoqi Li 和 Xiaosong Zhang。节省智能合约使用成本。在 *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03*, 2018，页码 81–84，2018。12. Bo Gao、Siyuan Shen、Ling Shi、Jiaying Li、Jun Sun 和 Lei Bu。智能合约的验证辅助 Gas 减少。在 *28th Asia-Pacific Software Engineering Conference APSEC 2021 Taipei Taiwan December 6-9 2021* 264 274 ，*Proceedings of the 41st International Conference, on Software Engineering, ICSE 2019, Montreal*，页码–。IEEE，2021。13. Neville Grech、Lexi Brent、Bernhard Scholz 和 Yannis Smaragdakis。Gigahorse：智能合约的全面声明式反编译。在 Joanne M. Atlee、Tevfik Bultan 和 Jon Whittle 编，*Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31*, 2019，页码 1176–1186。IEEE / ACM，2019。14. Neville Grech、Sifis Lagouvardos、Ilias Tsatiris 和 Yannis Smaragdakis。Elipmoc：以太坊智能合约的高级反编译。 *Proc. ACM Program. Lang.*, 6 77 1 77 27，2022（OOPSLA）：：–：。15. ´Akos Hajdu 和 Dejan Jovanovic。适用于SMT的 Solidity 内存模型形式化。在 Peter M¨uller 编，*Programming Languages and Systems - 29thEuropean Symposium on Programming, ESOP 2020 Held as Part of the EuropeanJoint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland April 25-30 2020 Proceedings* 12075 *Lecture Notes in Computer Science* 224 250 2020 ，卷，页码–。施普林格。16. Sifis Lagouvardos、Neville Grech、Ilias Tsatiris 和 Yannis Smaragdakis。以太坊“内存”的精确静态建模。*Proc. ACM Program. Lang.*，4 190 1 190 26，2020（OOPSLA）：：–：。17. Mayukh Mukhopadhyay。 *Ethereum Smart Contract Development*。Packt 出版社，2018。18. Julian Nagele 和 Maria A Schett，区块链超优化器，在 *Proceedings of 29th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2019)* 2019 ，。19. F. Nielson、H. R. Nielson 和 C. Hankin。 *Principles of Program Analysis*。施普林格，1999。20. Maria A. Schett 和 Julian Nagele。填充智能合约编译器的窥孔优化器。在 Bruno Bernardo 和 Diego Marmsoler 编， *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)* 84 *OpenAccess Series in Informatics (OASIcs)* 3 1 3 15 ，卷，页码：–：。Schloss Dagstuhl–Leibniz-Zentrum f¨ur Informatik，2020。21. Clara Schneidewind、Ilya Grishchenko、Markus Scherer 和 Matteo Maffei。eThor：以太坊智能合约的实用且可证明可靠的静态分析。在

Jay Ligatti、Xinming Ou、Jonathan Katz 和 Giovanni Vigna（编），*CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*、*USA*、*November 9-13 2020* 621 640 2020 ，第 – 页。ACM，。22. Petar Tsankov、Andrei Marian Dan、Dana Drachsler-Cohen、Arthur Gervais、Flo- rian B¨unzli 和 Martin T. Vechev。《Securify：智能合约的实用安全分析》。载于 David Lie、Mohammad Mannan、Michael Backes 和 XiaoFeng Wang（编），*Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*、*CCS 2018*、*Toronto*、*ON*、*Canada*、*October 15-19*，2018 年，第 67–82 页。ACM，2018 年。23. Gavin Wood。《以太坊：一种安全的去中心化通用交易账本》，2019 年。

# A 证明

## A.1 具体槽与健全抽象

我们以 Solidity 编译器生成代码的相关说明作为前提：
https://docs.soliditylang.org/en/v0.8.17/internals/layout in memory.html其主要要点如下：

1. 所有内存分配都采用相同的模式：先执行一条或多条指令 `MLOAD 0x40` 来读取空闲内存指针，随后执行一系列指令，并以属于集合 $\mathcal{K} = \{$`MSTORE 0x40`, `RETURN`, `REVERT`, `STOP`, `SELFDESTRUCT`$\}$ 的指令结束。2. 在 EVM 程序的任何执行过程中，空闲内存指针的值都会不断增大。3. 所有大于 `0x60` 的内存位置，都是通过使用地址为`0x40` 的空闲内存指针，并加上一个可选的、非负的偏移量来计算内存地址而访问的。该偏移量始终小于槽长度。

具体槽通过读取空闲内存指针以获取其基准引用的操作来标识，即执行轨迹中包含`MLOAD 0x40` 指令且读取相同空闲内存指针值的程序点子序列。具体槽由这些指令对应的程序点集合进行抽象。

分析的起点是一个完整且可靠的栈敏感控制流图（CFG），它使我们能够表示给定 EVM 程序的所有可能执行路径。上下文敏感性通过克隆从不同上下文（执行栈的状态）到达的基本块来实现。更多详情请参见 [7]。

现在，让我们形式化定义具体槽这一概念。
给定一条执行轨迹 $t \equiv t_0 \rightarrow^* t_n$，我们用 $inst(t_k)$表示在 $t_k$执行的、形式为 $pp{:}I$ 的指令；用 $stack(t_k, pos)$表示执行 $t_k$后栈中位置 $pos$ 的内容。我们用 $stack(t_k, top)$表示栈顶，用 $stack(t_k, top-1)$表示栈顶下方的元素，以此类推。最后，我们用 $freeptr(t_k)$表示在 $t_k$处执行指令后空闲内存指针的值。我们认为，当 $inst(t_k)$为$\in$ {`RETURN`、`REVERT`、`STOP`、`SELFDESTRUCT`}（即任何会终止 EVM 程序执行的指令）时，$freeptr(t_k)$返回 $-1$。

定义 7（具体槽）。考虑程序 $P$ 以及从初始状态 $t_0$ 开始的具体执行轨迹，其形式为 $t \equiv t_0 \rightarrow^* t_n$，其中 $n \geq 0$。我们将 $slots(t)$ 定义为在 $t$ 中执行 *`MLOAD 0x40`* 指令所得的不同值 $\{s_0, \ldots, s_k\}$ 的集合：

$$slots(t) = \{s | s \equiv stack(t_i, top) \wedge t_i \in t \wedge t_i \equiv pp{:}\textit{MLOAD 0x40}\}.$$

根据假设1 和2，每当一个临时槽被设为永久槽时，空闲内存指针就会向前推进，并为 $mem\langle 0x40 \rangle$赋予一个不同且更大的值。

定义 8（槽加载指令）。给定一个具有 $n \geq 0$ 的具体执行轨迹$t \equiv t_0 \rightarrow^* t_n$ 以及一个具体槽 $s \in slots(t)$，我们将 $t$ 中 $s$ 的加载指令定义为由以下内容定义的程序点多重集

$$loading(s) = \{pp \mid t_k \in t \wedge inst(t_k) \equiv pp{:}\texttt{MLOAD 0x40} \wedge stack(t_k, top) \equiv s\}.$$

定义 9（抽象槽位）。给定一条从初始状态 $t_0$ 出发、形式为 $t \equiv t_0 \rightarrow^* t_n$、具有 $n \geq 0$ 的具体执行轨迹，以及一个槽位 $s_i$，我们定义 $\alpha(s) = \{pp \mid pp \in loading(s)\}$。

## A.2 槽分析可靠性的证明

定义 10（EVM 分析方程组）。给定程序 $P$、其栈敏感控制流图 $CFG$（其中所有跳转指令均标记了其可能的目标基本块），以及形式为 $\rho(b, AS)$ 的传递函数，其中 $b$ 是一条以太坊虚拟机指令，$AS$ 是一个抽象状态。分析方程组 $\mathcal{E}(P)$ 包含以下方程：

| $pp{:}I$ | 方程 | |
|---|---|---|
| (1) `JUMP/I DEST` | $\mathcal{X}_d \sqsupseteq \mathcal{X}_{pp-1}$ | $\forall\ d \in$ `DEST` |
| (2) $otherwise$ | $\mathcal{X}_{pp} \sqsupseteq \rho(I, \mathcal{X}_{pp-1})$ | |

方程（1）中的目标集合 `DEST` 包含一个或两个目标程序点，具体取决于 $pp$ 处的指令。

求解约束方程组时，约束变量会对程序的指向信息进行过度近似。特别地，形式为 $\mathcal{X}_{pp}$ 的变量存储了在 $pp$ 处的指令执行后所得到的槽信息。此类方程组可以通过迭代求解。一种朴素算法首先将每个约束变量初始化为 $\{\emptyset\}$，然后按如下方式迭代细化这些变量的值：

1. 将约束变量的当前值代入每个约束的右侧，并在必要时计算右侧的值；2. 如果每个约束 $\mathcal{X} \sqsupseteq E$ 都成立，其中 $E$ 是上一步计算得到的右侧的值，则流程结束；否则，3. 对于每个不成立的 $\mathcal{X} \sqsupseteq E$，令 $E'$ 为 $\mathcal{X}$ 的当前值。然后将 $\mathcal{X}$ 的当前值更新为 $E \sqcup E'$。迭代应用完所有这些更新后，我们在步骤 1 重复此过程。

由于分析中使用的抽象域不存在无限上升链，因此可以保证终止。

定理 3（槽位分析的可靠性）。考虑一个程序 $P$，以及为 $P$ 计算得出的已分配抽象槽位集合 $\mathcal{S}_{all}$。对于任意执行轨迹 $t \equiv t_0 \rightarrow^* t_n \in P$，对于所有 $s \in slots(t)$，都存在 $a \in \mathcal{S}_{all}$ 使得 $\alpha(s) \subseteq a$。

槽位分析结果收集于 $\mathcal{S}_{all}$ 的某些程序点上，具体来说，是 $\mathcal{K}$中各指令之前的程序点。我们基于以下引理证明定理3 ，该引理指出 $\mathcal{S}_{pp}$ 是由空闲内存指针指向的临时槽的可靠过近似。

引理 1. 考虑一个程序 $P$ 和 $\mathcal{X}_{pp}$ 对于每个程序点 $pp$在 $P$中，使用传递函数 *10* 得到定义 10 的方程组的一个解 $\nu$。对于任意执行轨迹 $t \equiv t_0 \rightarrow^* t_n \in P$，我们有，对于每个 $t_k$，$0 \leq k \leq n$，其中$s \equiv freeptr(t_k)$ 且 $pp$：$I \equiv inst(t_k)$，存在 $a \in \mathcal{X}_{pp}$ 使得 $\alpha(s) \subseteq a$。

证明。我们可以对 $n$ 的值（即轨迹$t_0 \rightarrow^* t_n$的长度）进行归纳推理。

基础情形： 如果 $n = 0$ 那么 $slots(t)$ 为空，引理 1 显然成立。

归纳假设： 我们假设引理 1 对长度为 $m \leq n$ 且满足 $n \geq 0$的所有轨迹均成立：对于任意轨迹 $t_0 \rightarrow^m t_m$，满足 $inst(t_m) \equiv pp$：$I$且 $freeptr(t_m) \equiv s$，存在 $a \in \mathcal{X}_{pp}$ 使得 $\alpha(s) \subseteq a$。

归纳情形：考虑长度为 $t_0 \rightarrow \cdots \rightarrow t_n \rightarrow t_{n+1}$的执行轨迹 $n+1 >$0。为将定理推广到长度为 $n+1$ 的执行轨迹，我们针对定义 $\nu$ 1 中的传递函数的所有可能情形进行推理，如下所示。

（1） $inst\ (t_{n+1}) \equiv pp$：MLOAD 0x40。

在此情形下，所考虑的方程为

$$\mathcal{X}_{pp} \sqsupseteq \{s \cup \{pp\} \mid s \in \mathcal{X}_{pp-1}\} \tag{1}$$

其中， $pp-1$ 是步骤 $t_n$执行的指令所在的程序点。
根据归纳假设，给定 $s \equiv freeptr(t_n)$，在步骤 $t_n$ 存在一个集合 $a \in \mathcal{X}_{pp-1}$ 使得 $\alpha(s) \subseteq a$。由于在 $t_{n+1}$执行的指令是 `MLOAD 0x40`，空闲内存指针相对于 $t_n$ 保持不变，并且根据定义 9， $\alpha(s)$将包含截至$t_n$所累积的程序点，以及程序点 $pp$。根据方程 1，不难看出$\alpha(s) \subseteq \mathcal{X}_{pp}$ 成立。

(2) $inst(t_{n+1}) \in \mathcal{K}$。

在此情况下，所考虑的等式为

$$\mathcal{X}_{pp} \sqsupseteq \{\emptyset\} \tag{2}$$

根据在 $t_{n+1}$ 处执行的指令，分为两种子情况：

- $inst(t_{n+1}) \equiv$ `MSTORE 0x40`。根据本节开头详述的假设 2，在 EVM 程序的任何一次执行中，空闲内存指针都会被赋予一个持续递增的值。这意味着，$t_0 \rightarrow^{n+1} t_{n+1}$ 中的任何`MLOAD 0x40` 指令都无法加载值$s \equiv freeptr(t_{n+1})$，因此 $\alpha(s)$为空，且 $\alpha(s) \subseteq \mathcal{X}_{pp}$ 显然成立。
- $inst(t_{n+1})$是集合{`RETURN`、`REVERT`、`STOP`、`SELFDESTRUCT`}中的一条执行终止指令。在这种情况下， $s \equiv freeptr(t_{n+1})$返回 $-1$， $\alpha(s)$也为空，且 $\alpha(s) \subseteq \mathcal{X}_{pp}$。

(3) $inst(t_{n+1}) \notin \mathcal{K} \cup \{\texttt{MLOAD 0x40}\}$。引理 1 在这种情况下显然成立，因为所应用的等式只是在任何执行轨迹中传播前一条已执行指令的抽象状态。

定理3 的证明思路如下。我们用 $pred(pp)$表示在执行轨迹中可能先于 $pp$ 出现的程序点集合。

– 我们在抽象域上定义关系算子 $\preceq$ 如下：当且仅当对所有 $a \in A$，都存在 $b \in B$ 使得 $a \subseteq b$，$A \preceq B$成立。– 给定定义 10 中方程组的一个解，很容易证明，对于每个满足 $I \neq \texttt{MSTORE 0x40}$ 的程序点 $pp{:}I \in P$，$\mathcal{X}_{pp'} \preceq \mathcal{X}_{pp}$（其中 $pp' \in pred(pp)$）成立。– $\mathcal{S}_{all}$ 定义为 $\bigcup_{pp \in \mathcal{P}_W} \mathcal{X}_{pp-1}$，其中 $\mathcal{P}_W$ 是所有满足 $I \in \mathcal{K}$ 的程序点$pp{:}I$ 的集合，它收集了包含最大程序点集合的抽象状态。– 因此，对于每个程序点 $pp{:}I \in P$，$\mathcal{X}_{pp} \preceq \mathcal{S}_{all}$ 成立。

显然可以证明，定理 3 由最后一点和引理 1即可得出。

### A.3 槽访问分析的可靠性证明

我们首先定义要近似的性质，即具体槽访问的概念：

定义 11（具体槽访问）。给定一个带有 $n \geq 0$ 的具体执行轨迹$t \equiv t_0 \rightarrow^* t_n$ 、一个形如 $inst(t_k) \equiv pp{:}I$ 的步骤以及一个具体槽$s$。我们定义 $is_accesed(s, pp, pos)$，当且仅当在程序点 $pp$ 通过栈位置 $pos$ 访问了 $s$ 时，该函数返回 true。我们定义

$$slots_accessed(pp, pos) = \{s \mid t_k \in t \wedge inst(t_k) \equiv pp{:}I \wedge is_accessed(s, pp, pos)\}$$

槽访问分析首先求解定义 10 中定义的方程组，其中抽象状态 AS 是内存分析抽象状态，该状态定义于定义 3 ，且 $\rho$ 是内存分析传递函数 $\tau$ ，该函数定义于定义 4。

方程组的最小解是对程序点处所访问具体槽位的安全近似，我们通过以下定理陈述这一点。不失一般性，我们假设分析返回一个映射 $\pi(x)$，其中存储的是包含于 $\mathcal{S}_{all}$ 中的程序点集合，而非标识符集合。

定理 4（槽访问分析的可靠性）。给定任意具体执行轨迹 $t \equiv t_0 \rightarrow^* t_n$、一个形式为 $inst(t_k) \equiv pp{:}I$ 的步骤、一个栈位置 $pos$、一个具体槽 $s$ ，满足 $s \in slots_accessed(pp, pos)$，以及槽访问分析的结果 $\pi$，则存在 $a \in \mathcal{X}_{pred(pp)}(pos)$，使得 $\alpha(s) \subseteq a$。

证明。我们可以对 $n$ 的值进行归纳，即对轨迹$t_0 \to^* t_n$的长度进行归纳。

基础情形：如果 $n = 0$，那么 $slots_accessed(,)$ 为空，该定理显然成立。

归纳假设：对于任意满足 $inst(t_m) \equiv pp{:}I$ 的执行轨迹 $t_0 \to^m t_m$，给定使用 $\tau$ 作为传递函数的方程组的解，我们有 $\mathcal{X}_{pp}$ 能够正确跟踪所有潜在的抽象槽位

归纳情形：考虑长度为 $t_0 \to \cdots \to t_n \to t_{n+1}$ 的执行轨迹 $n+1 >0$。为将定理扩展到长度为 $n+1$ 的执行轨迹，我们针对定义 $\tau$ 4 中传递函数 的所有可能情形进行如下推理。

(1) $inst(t_{n+1}) \equiv pp{:}$`MLOAD 0x40`。根据定理 3 的可靠性，由 get slots($pp$) 得到的抽象槽位是具体槽位的可靠过近似。(2) $inst(t_{n+1}) \equiv pp{:}$`MLOAD`。 -

在具体轨迹中，一次内存访问可能读取两种潜在类型的值：

- 与某个存储槽的基准引用相对应的值。传递函数会按如下方式更新抽象状态的栈顶：$\pi[t \mapsto \{m \mid s \in \pi(t) \wedge m \in \pi(s)\}]$，它会将执行该指令前位于栈顶的所有可用存储槽中 $t$ 可到达的所有存储槽 $m$ 保留在栈顶，MLOAD 指令。
- 通用值：操作数直接从栈中弹出。

(3) $inst(t_{n+1}) \equiv pp$：`MSTORE`
在具体轨迹中，一次内存访问可能会写入两种可能类型的值：

- 与存储槽基准引用相对应的值。在这种情况下，传递函数按如下方式更新抽象状态：$\pi[s \mapsto \pi(s) \cup \pi(t-1)] \setminus \{t, t-1\} \forall s \in \pi(t)$，即使用栈顶下方一个元素中所有可用的基准引用，更新栈顶所有可用存储槽的条目
- 通用值：传递函数只需弹出栈顶的两个元素。

(4) $inst(t_{n+1}) \equiv pp$：`SWAPi`。它显然成立，因为它只是交换抽象状态中的两个条目。(5) $inst(t_{n+1}) \equiv pp$：`DUPi`。它显然成立，因为它只是复制抽象状态中的两个条目。(6) 其他情况。它显然成立，因为它只移除相应指令从栈中弹出的那些条目。

鉴于前两个定理的可靠性，我们直接证明最后一个推论的可靠性。

推论 2（不必要的写访问推断的可靠性）。如果 $pp \in \mathcal{N}$，则不存在这样的程序执行：在执行 $pp$ 处的写指令后，读取 $pp$ 处访问的存储槽。

证明。根据控制流图的可靠性和定理4，显然成立。

此图“running2-cfg.png”提供 PNG 格式版本，下载地址：

http://arxiv.org/ps/2301.04757v1