零知识证明的先进形式化验证：如何证明零知识内存？

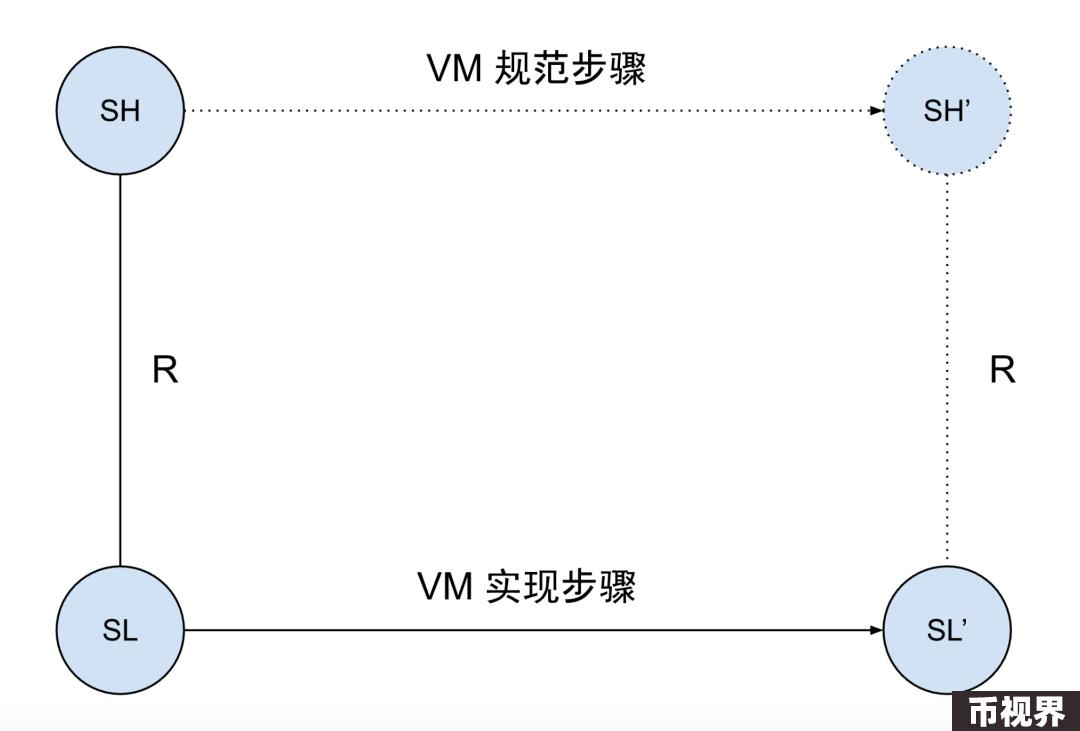
在关于零知识证明的先进形式化验证的系列文章中，我们已经讨论了如何验证ZK指令以及对两个ZK漏洞的深度剖析。

正如在公开报告（https://skynet.certik.com/projects/zkwasm）和代码库（https://github.com/CertiKProject/zkwasm-fv）中所显示的，通过形式化验证每一条zkWasm指令，我们找到并修复了每一个漏洞，从而能够完全验证整个zkWasm电路的技术安全性和正确性。

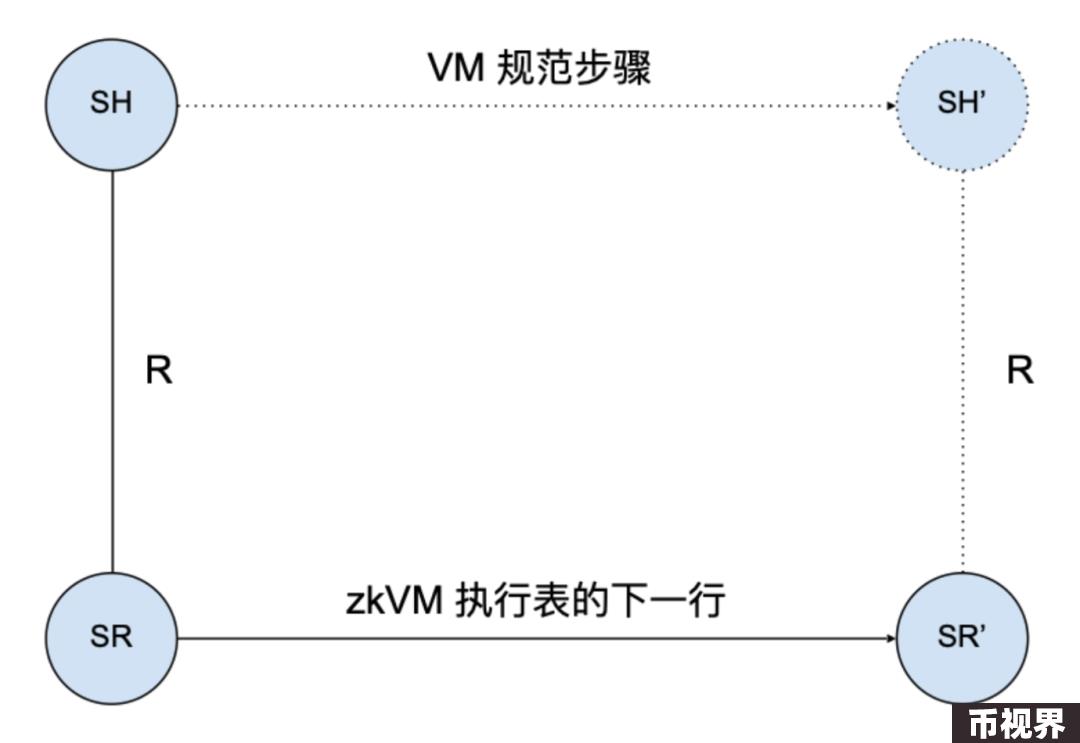
尽管我们已展示了验证一条zkWasm指令的过程，并介绍了相关的项目初步概念，但熟悉形式化验证读者可能更想了解zkVM与其他较小的ZK系统、或其他类型的字节码VM在验证上的独特之处。在本文中，我们将深入讨论在验证zkWasm内存子系统时所遇到的一些技术要点。内存是zkVM最为独特的部分，处理好这一点对所有其他zkVM的验证都至关重要。

### 形式化验证：虚拟机（VM）对 ZK虚拟机（zkVM）

我们的最终目标是验证zkWasm的正确性，其与普通的字节码解释器（VM，例如以太坊节点所使用的EVM解释器）的正确性定理相似。亦即，解释器的每一执行步骤都与基于该语言操作语义的合法步骤相对应。如下图所示，如果字节码解释器的数据结构当前状态为SL，且该状态在Wasm机器的高级规范中被标记为状态SH，那么当解释器步进到状态SL'时，必须存在一个对应的合法高级状态SH'，且Wasm规范中规定了SH必须步进到SH'。



同样地，zkVM也有一个类似的正确性定理：zkWasm执行表中新的每一行都与一个基于该语言操作语义的合法步骤相对应。如下图所示，如果执行表中某行数据结构的当前状态是SR，且该状态在Wasm机器的高级规范中表示为状态SH，那么执行表的下一行状态SR'必须对应一个高级状态SH'，且Wasm规范中规定了SH必须步进到SH'。



由此可见，无论是在VM还是zkVM中，高级状态和Wasm步骤的规范是一致的，因此可以借鉴先前对编程语言解释器或编译器的验证经验。而zkVM验证的特殊之处在于其构成系统低级状态的数据结构类型。

首先，如我们在之前的文章中所述，zk证明器在本质上是对大素数取模的整数运算，而Wasm规范和普通解释器处理的是32位或64位整数。zkVM实现的大部分内容都涉及到此，因此，在验证中也需要做相应的处理。然而，这是一个“本地局部”问题：因为需要处理算术运算，每行代码变得更复杂，但代码和证明的整体结构并没有改变。

另一个主要的区别是如何处理动态大小的数据结构。在常规的字节码解释器中，内存、数据栈和调用栈都被实现为可变数据结构，同样的，Wasm规范将内存表示为具有get/set方法的数据类型。例如，Geth的EVM解释器有一个Memory数据类型，它被实现为表示物理内存的字节数组，并通过Set32和GetPtr方法写入和读取。为了实现一条内存存储指令，Geth调用Set32来修改物理内存。

func opMstore(pc \*uint64, interpreter \*EVMInterpreter, scope \*ScopeContext) ([]byte, error) { // pop value of the stack mStart, val := scope.Stack.pop(), scope.Stack.pop() scope.Memory.Set32(mStart.Uint64(), &val) return nil, nil}

在上述解释器的正确性证明中，我们在对解释器中的具体内存和在规范中的抽象内存进行赋值之后，证明其高级状态和低级状态相互匹配，这相对来说是比较容易的。

然而，对于zkVM而言，情况将变得更加复杂。

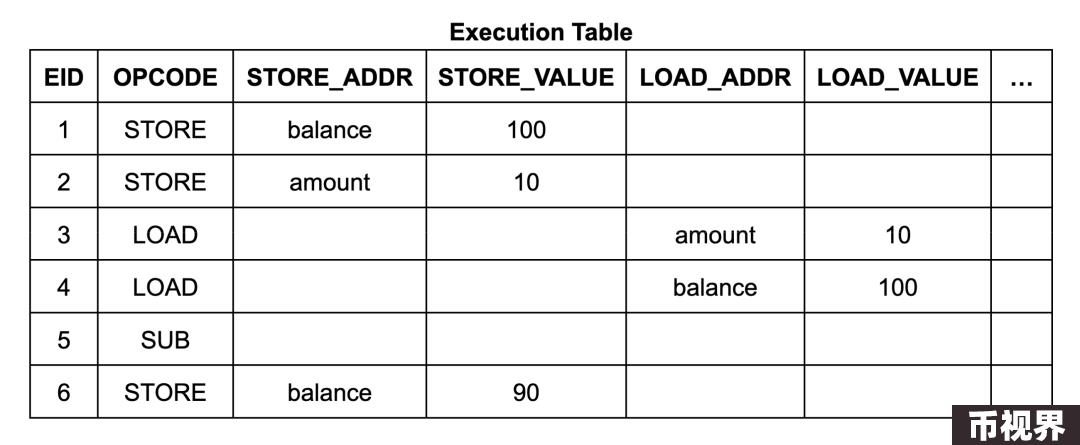
### zkWasm的内存表和内存抽象层

在zkVM中，执行表上有用于固定大小数据的列（类似于CPU中的寄存器），但它不能用来处理动态大小的数据结构，这些数据结构要通过查找辅助表来实现。zkWasm的执行表有一个EID列，该列的取值为1、2、3……，并且有内存表和跳转表两个辅助表，分别用于表示内存数据和调用栈。

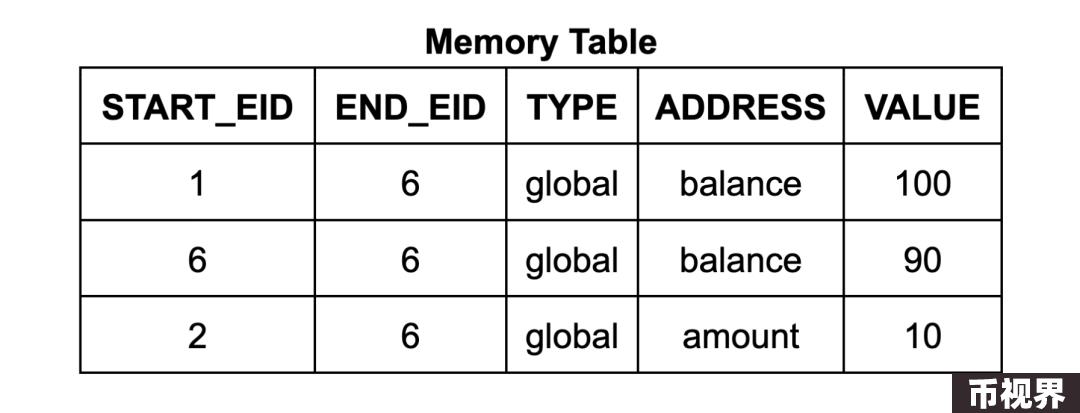
以下是一个提款程序的实现示例：

int balance, amount;void main () {balance = 100; amount = 10;balance -= amount; // withdraw}

执行表的内容和结构相当简单。它有6个执行步骤（EID1到6），每个步骤都有一行列出其操作码（opcode），如果该指令是内存读取或写入，则还会列出其地址和数据：



内存表中的每一行都包含地址、数据、起始EID和终止EID。起始EID是写入该数据到该地址的执行步骤的EID，终止EID是下一个将会写入该地址的执行步骤的EID。（它还包含一个计数，我们稍后详细讨论。）对于Wasm内存读取指令电路，其使用查找约束来确保表中存在一个合适的表项，使得读取指令的EID在起始到终止的范围内。（类似地，跳转表的每一行对应于调用栈的一帧，每行均标有创建它的调用指令步骤的EID。）



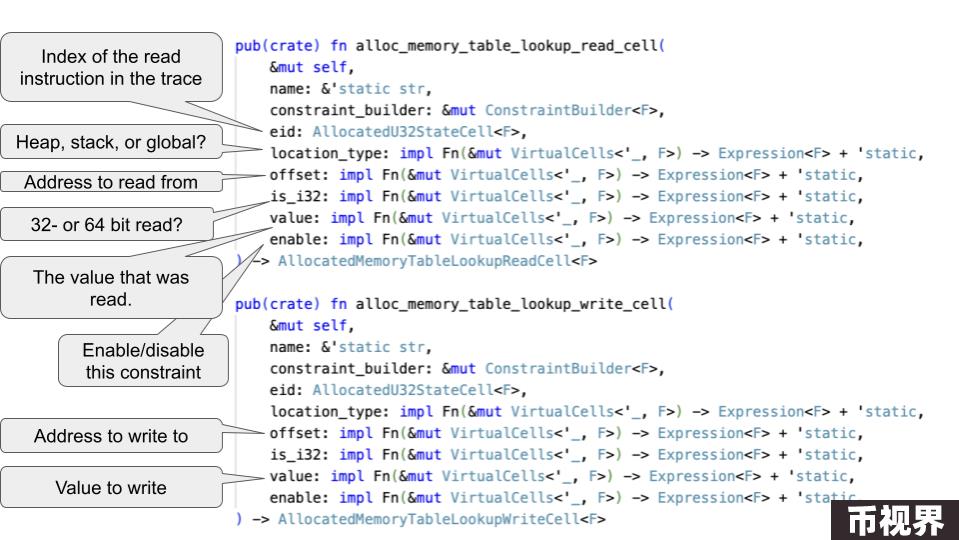
这个内存系统与常规VM解释器的区别很大：内存表不是逐步更新的可变内存，而是包含整个执行轨迹中所有内存访问的历史记录。为了简化程序员的工作，zkWasm提供了一个抽象层，通过两个便捷入口函数来实现。分别是：

alloc\_memory\_table\_lookup\_write\_cell

和

Alloc\_memory\_table\_lookup\_read\_cell

其参数如下：

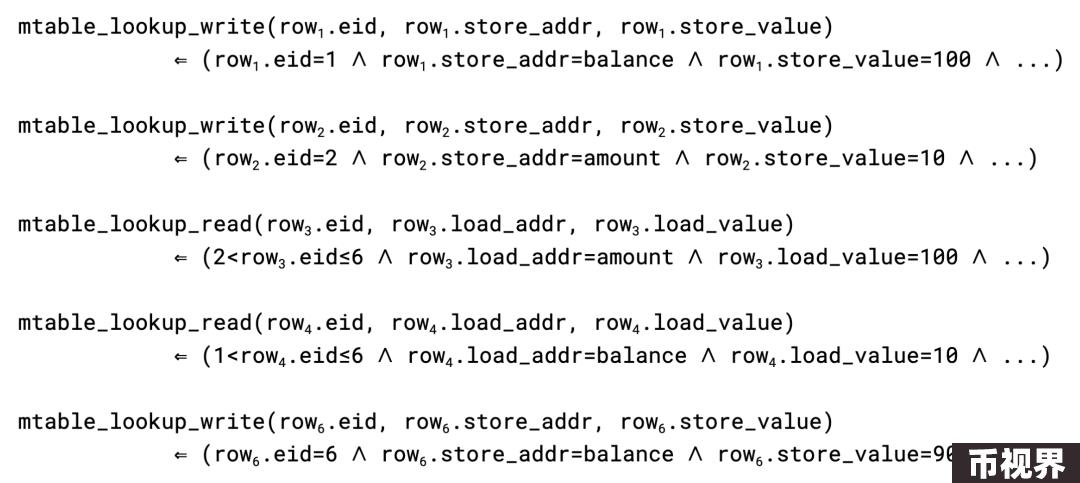


例如，zkWasm 中实现内存存储指令的代码包含了一次对write alloc函数的调用：

let memory\_table\_lookup\_heap\_write1 = allocator .alloc\_memory\_table\_lookup\_write\_cell\_with\_value( "store write res1", constraint\_builder, eid, move |\_\_\_\_| constant\_from!(LocationType::Heap as u64), move |meta| load\_block\_index.expr(meta), // address move |\_\_\_\_| constant\_from!(0), // is 32-bit move |\_\_\_\_| constant\_from!(1), // (always) enabled );letstore\_value\_in\_heap1=memory\_table\_lookup\_heap\_write1.value\_cell;

alloc函数负责处理表之间的查找约束以及将当前eid与内存表条目相关联的算术约束。由此，程序员可以将这些表看作普通内存，并且在代码执行之后store\_value\_in\_heap1的值已被赋给了load\_block\_index地址。

类似地，内存读取指令使用read\_alloc函数实现。在上面的示例执行序列中，每条加载指令有一个读取约束，每条存储指令有一个写入约束，每个约束都由内存表中的一个条目所满足。



形式化验证的结构应与被验证软件中所使用的抽象相对应，使得证明可以遵循与代码相同的逻辑。对于zkWasm，这意味着我们需要将内存表电路和“alloc read/write cell”函数作为一个模块来进行验证，其接口则像可变内存。给定这样的接口后，每条指令电路的验证可以以类似于常规解释器的方式进行，而额外的ZK复杂性则被封装在内存子系统模块中。

在验证中，我们具体实现了“内存表其实可以被看作是一个可变数据结构”这个想法。亦即，编写函数memory\_at type，其完整扫描内存表、并构建相应的地址数据映射。（这里变量type的取值范围为三种不同类型的Wasm内存数据：堆、数据栈和全局变量。）而后，我们证明由alloc函数所生成的内存约束等价于使用set和get函数对相应地址数据映射所进行的数据变更。我们可以证明：

memory\_table\_lookup\_read\_celleidtypeoffsetvalue

则

get(memory\_ateidtype)offset=Somevalue

memory\_table\_lookup\_write\_cell eid type offset value

则

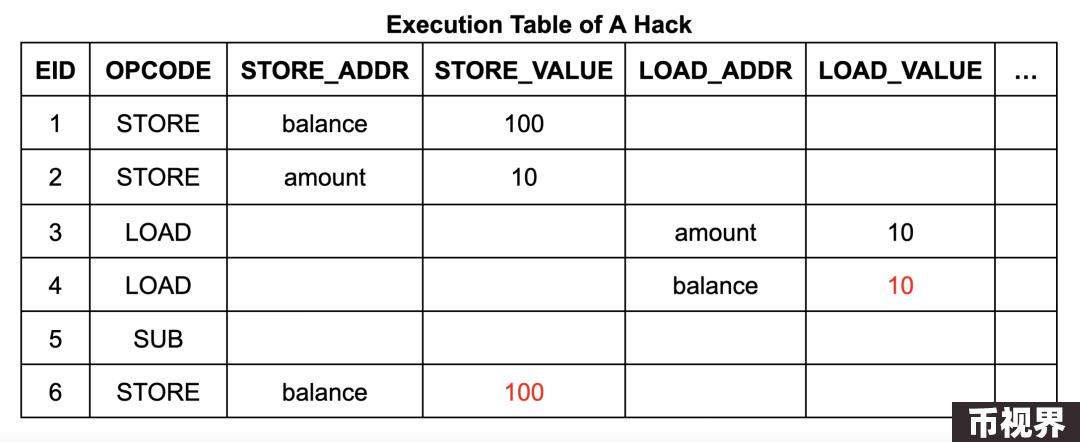
memory\_at (eid+1) type = set (memory\_at eid type) offset value

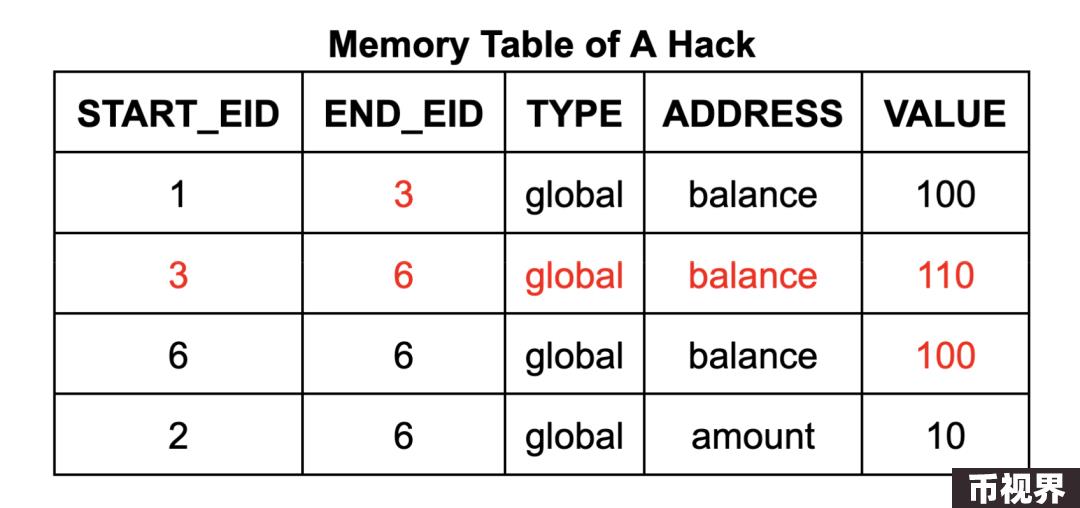
在此之后，每条指令的验证可以建立在对地址数据映射的get和set操作之上，这与非ZK字节码解释器相类似。

### zkWasm的内存写入计数机制

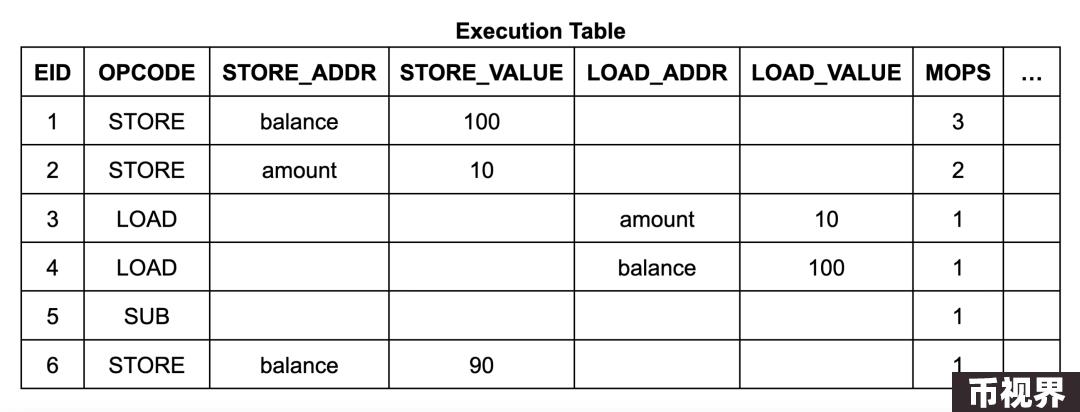
不过，上述的简化描述并未揭示内存表和跳转表的全部内容。在zkVM的框架下，这些表可能会受到攻击者的操控，攻击者可以轻易地通过插入一行数据来操纵内存加载指令，返回任意数值。

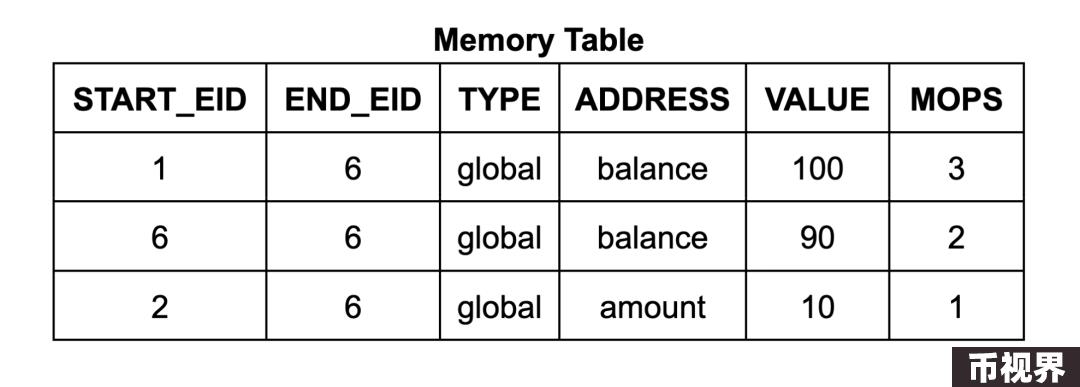
以提款程序为例，攻击者有机会在提款操作前，通过伪造一个$110的内存写入操作，将虚假数据注入到账户余额中。这一过程可以通过在内存表中添加一行数据，并修改内存表和执行表中现有单元格的数值来实现。这将导致其可以进行“免费”的提款操作，因为账户余额在操作后将仍然保持在$100。





为确保内存表（和跳转表）仅包含由实际执行的内存写入（和调用及返回）指令生成的有效条目，zkWasm采用了一种特殊的计数机制来监控条目数量。具体来说，内存表设有一个专门的列，用以持续追踪内存写入条目的总数。同时，执行表中也包含了一个计数器，用于统计每个指令预期进行的内存写入操作的次数。通过设置一个相等约束，从而确保这两个计数是一致的。这种方法的逻辑十分直观：每当内存进行写入操作，就会被计数一次，而内存表中相应地也应有一条记录。因此，攻击者无法在内存表中插入任何额外的条目。





上面的逻辑陈述有点模糊，在机械化证明的过程中，需要使其更加精确。首先，我们需要修正前述内存写入引理的陈述。我们定义函数mops\_at eid type，对具有给定eid和type的内存表条目计数（大多数指令将在一个eid处创建0或1个条目）。该定理的完整陈述有一个额外的前提条件，指出没有虚假的内存表条目：

如果以下约束成立

(memory\_table\_lookup\_write\_celleidtypeoffsetvalue)

并且以下新增约束成立

(mops\_ateidtype)=1

则

(memory\_at(eid+1)type)=set(memory\_ateidtype)offsetvalue

这要求我们的验证比前述情况更精确。 仅仅从相等约束条件中得出内存表条目总数等于执行中的总内存写入次数并不足以完成验证。为了证明指令的正确性，我们需要知道每条指令对应了正确数目的内存表条目。例如，我们需要排除攻击者是否可能在执行序列中略去某条指令的内存表条目，并为另一条无关指令创建一个恶意的新内存表条目。

为了证明这一点，我们采用了由上至下的方式，对给定指令对应的内存表条目数量进行限制，这包括了三个步骤。首先，我们根据指令类型为执行序列中的指令预估了所应该创建的条目数量。我们称从第 i 个步骤到执行结束的预期写入次数为instructions\_mops i，并称从第 i 条指令到执行结束在内存表中的相应条目数为cum\_mops (eid i)。通过分析每条指令的查找约束，我们可以证明其所创建的条目不少于预期，从而可以得出所跟踪的每一段 [i ... numRows] 所创建的条目不少于预期：

Lemma cum\_mops\_bound': forall n i, 0 ≤ i -> i + Z.of\_nat n = etable\_numRow ->MTable.cum\_mops(etable\_valueseid\_celli)(max\_eid+1)≥instructions\_mops'in.

其次，如果能证明表中的条目数不多于预期，那么它就恰好具有正确数量的条目，而这一点是显而易见的。

Lemma cum\_mops\_equal' : forall n i, 0 ≤ i -> i + Z.of\_nat n = etable\_numRow -> MTable.cum\_mops (etable\_values eid\_cell i) (max\_eid+1) ≤ instructions\_mops' i n ->MTable.cum\_mops(etable\_valueseid\_celli)(max\_eid+1)=instructions\_mops'in.

现在进行第三步。我们的正确性定理声明：对于任意n，cum\_mops和instructions\_mops在表中从第n行到末尾的部分总是一致的：

Lemma cum\_mops\_equal : forall n, 0 <= Z.of\_nat n < etable\_numRow ->MTable.cum\_mops(etable\_valueseid\_cell(Z.of\_natn))(max\_eid+1)=instructions\_mops(Z.of\_natn)

通过对n进行归纳总结来完成验证。表中的第一行是zkWasm的等式约束，表明内存表中条目的总数是正确的，即cum\_mops 0 = instructions\_mops 0。对于接下来的行，归纳假设告诉我们：

cum\_mopsn=instructions\_mopsn

并且我们希望证明

cum\_mops (n+1) = instructions\_mops (n+1)

注意此处

cum\_mops n = mop\_at n + cum\_mops (n+1)

并且

instructions\_mops n = instruction\_mops n + instructions\_mops (n+1)

因此，我们可以得到

mops\_at n + cum\_mops (n+1) = instruction\_mops n + instructions\_mops (n+1)

此前，我们已经证明了每条指令将创造不少于预期数量的条目，例如

mops\_at n ≥ instruction\_mops n.

所以可以得出

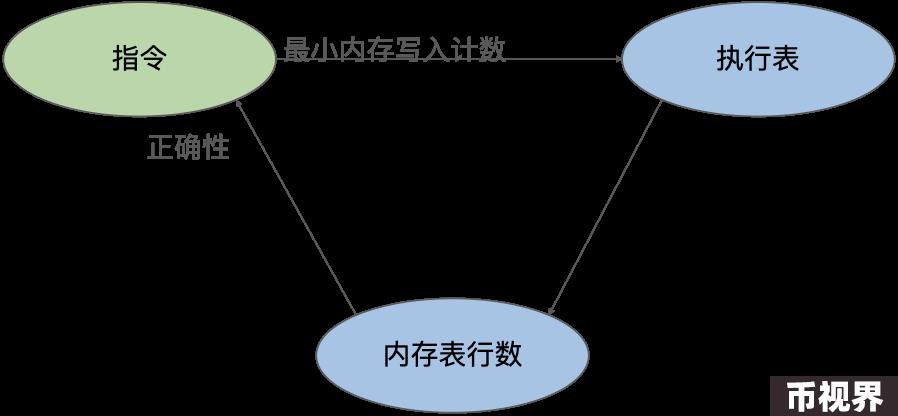
cum\_mops (n+1) ≤ instructions\_mops (n+1)

这里我们需要应用上述第二个引理。

如此详细地说明证明过程，是形式化验证的典型特征，也是验证特定代码片段通常比编写它需要更长时间的原因。然而这样做是否值得？在这里的情况下是值得的，因为我们在证明的过程中的确发现了一个跳转表计数机制的关键错误。之前的文章中已经详细描述了这个错误——总结来说，旧版本的代码同时计入了调用和返回指令，而攻击者可以通过在执行序列中添加额外的返回指令，来为伪造的跳转表条目腾出空间。尽管不正确的计数机制可以满足对每条调用和返回指令都计数的直觉，但当我们试图将这种直觉细化为更精确的定理陈述时，问题就会凸显出来。

### 使证明过程模块化

从上面的讨论中，我们可以看到在关于每条指令电路的证明和关于执行表的计数列的证明之间存在着一种循环依赖关系。要证明指令电路的正确性，我们需要对其中的内存写入进行推理；即需要知道在特定EID处内存表条目的数量、以及需要证明执行表中的内存写入操作计数是正确的；而这又需要证明每条指令至少执行了最少数量的内存写入操作。



此外，还有一个需要考虑的因素，zkWasm项目相当庞大，因此验证工作需要模块化，以便多位验证工程师分工处理。因此，对计数机制的证明解构时需要特别注意其复杂性。例如，对于LocalGet指令，有两个定理如下：

Theorem opcode\_mops\_correct\_local\_get : forall i, 0 <= i -> etable\_values eid\_cell i > 0 -> opcode\_mops\_correct LocalGet i.

Theorem LocalGetOp\_correct : forall i st y xs, 0 <= i -> etable\_values enabled\_cell i = 1 -> mops\_at\_correct i -> etable\_values (ops\_cell LocalGet) i = 1 -> state\_rel i st -> wasm\_stack st = xs -> (etable\_values offset\_cell i) > 1 -> nth\_error xs (Z.to\_nat (etable\_values offset\_cell i - 1)) = Some y ->state\_rel(i+1)(update\_stack(incr\_iidst)(y::xs)).

第一个定理声明

opcode\_mops\_correct LocalGet i

展开定义后，意味着该指令在第i行至少创建了一个内存表条目（数字1是在zkWasm的LocalGet操作码规范中指定的）。

第二个定理是该指令的完整正确性定理，它引用

mops\_at\_correct i

作为假设，这意味着该指令准确地创建了一个内存表条目。

验证工程师可以分别独立地证明这两个定理，然后将它们与关于执行表的证明结合起来，从而证得整个系统的正确性。值得注意的是，所有针对单个指令的证明都可以在读取/写入约束的层面上进行，而无须了解内存表的具体实现。因此，项目分为三个可以独立处理的部分。



### 总结

逐行验证zkVM的电路与验证其他领域的ZK应用并没有本质区别，因为它们都需要对算术约束进行类似的推理。从高层来看，对zkVM的验证需要用到许多运用于编程语言解释器和编译器形式化验证的方法。这里主要的区别在于动态大小的虚拟机状态。然而，通过精心构建验证结构来匹配实现中所使用的抽象层，这些差异的影响可以被最小化，从而使得每条指令都可以像对常规解释器那样，基于get-set接口来进行独立的模块化验证。