Disclaimer: For accuracy, the original English version of Cartesi Core's Techpaper must be referred to.

Cartesi白皮书

摘要

Cartesi是用于开发和部署可扩展DApps（分布式应用）的第2层平台*（"位于底层链和DApp之间，通过高性能的链下计算，将Layer 1从执行合约和计算的海量工作中解放出来，专心处理token流转和计算结果存储，从而提升整体可扩展性"）*。 Cartesi DApps由链上及链下两部分模块组成。链下模块在Cartesi节点内运行，代表了每个DApp用户的权益。 Cartesi节点为DApp开发者提供可重现的Cartesi Machine，可以运行大规模可信计算。这些可信计算很容易通过强大的原语集成到智能合约中，这些原语提供了更大的灵活度和复杂逻辑处理能力。更准确地说，任何因Cartesi Machine内部计算产生有争议性的结果，均可在区块链上以相当低的成本进行仲裁*（"Gas费用奖善罚恶"）*。Cartesi节点还允许DApp开发者运行本地代码。本地计算更可利用节点的全部处理能力，其中包括任何可用的显卡算力。无论是由节点本地执行还是在Cartesi Machines系统内执行，链下模块均会在完整的"Linux"操作系统下运行*（"该操作系统提供复杂计算所需的完整生态系统"）*。Cartesi使DApp开发者能够使用他们已经熟悉的所有编程语言，工具，库，软件和服务。通过将其DApps的大多数复杂逻辑移动到便捷的链下模块，开发者可以摆脱区块链所固有的限制和特性。 通过这种方式，Cartesi允许开发者能够选择最佳的运行实例环境来托管其DApp的每个部分。

1 简介

公共区块链是网络可以通过其在共享状态下维持分散共识的机制。 通常，除了其他数据之外，该状态还包含支付系统。在由此产生的经济中，参与者所持有的股份是他们的动机，使国家广泛地向其他国家开放，并拒绝无效的交易。在这种良性循环中，支付系统建立在分散的共识之上，而分散共识的作用只取决于支付系统本身所创造的激励机制。然后，支付系统和共识都可以用于其他目的。

随着区块链技术的新应用的设想，对底层基础设施的需求不断增加。目前，广泛采用区块链技术的两大障碍是其可扩展性差，缺乏稳固的开发环境。 Cartesi对区块链生态系统的主要贡献是克服这两个问题。

可扩展性 目前部署的共识机制基于完全冗余[Nakamoto 2009; Wood 2018]。 它们要求每个交易都永久存储，并由每个参与者进行验证。这种低效率是交易率增长，涉及的数据量以及交易内计算强度的关键限制因素。 高交易成本和增加的延迟已成为许多创新应用的障碍，否则这些应用将受益于智能合约为区块链带来的灵活性。

尝试提高区块链可扩展性可分为第1层和第2层解决方案。第1层可扩展性解决方案改变了底层的区块链基础设施本身。 示例包括区块大小，分片和委托证明（DPoS）的优化。 因为它们在基础设施层面运行，所以这些解决方案受到保持全球共识的要求的负担。国家的某些方面，例如支付系统，对所有各方都至关重要，因此需要全球共识。否则，对于区块链调解的大多数交互，将访问和验证责任限制在可能受影响的少数方面是完全安全的。然后，区块链可用于提供最终结果，并在极少数情况下保证当地达成共识。 换句话说，全球共识是一种宝贵的资源，应该节俭的加以利用。 认识到这一事实，诸如等离子，侧链，TrueBit或状态通道的第2层可扩展性解决方案尽可能多地移动数据和计算。 在第2节中深入讨论了第1层和第2层可扩展性解决方案。

计算环境 无论在链上或是在链下，每当出现影响交易结果的计算执行时，都必须由所有参与验证的角色进行验证。可重现计算模型必须是自包含的和确定的，换句话说，必须完全规定并商定计算的完整状态和对该状态的整个顺序进行修改。但可悲的是，现存真正的计算体系结构并没有考虑到这些条件，因此不可重现。区块链平台通过在处理智能合约时使用自定义虚拟机（VMs）来解决此问题，这些VM是可重现的，但只存在于一些特定情况下。一方面，它们为对智能合约有用的功能，提供了本地支持（例如，交流计数，回滚，关联存储器，认证，密码等）。另一方面，它们缺乏通用体系结构中的有价值的特性（例如，浮点运算，虚拟内存，中断等）。

在过去几十年里，全球软件行业的变革，可归结于两个关键因素。 首先是现代硬件平台处理大量数据的速度呈指数级增长。第二个同样重要的是软件开发环境不断增强的表现力。 实际上，通用计算并非计算孤岛。 相反，它更依赖于全球软件开发者的通力协作，参与共建模块相互间的组合。这些模块和服务依赖于底层操作系统（内存管理，进程管理，文件系统，网络等）托管的标准库工具。它可理解为一个"将所有内容联系在一起的操作系统"。这些设施不能通过典型区块链为智能合约开发者提供的独立编程语言和编译器提供。

可重现性和可扩展性问题使得链上计算环境非常严格，所以，为了提高效率并扩大区块链开发的范围，我们需要一个支持现代操作系统的可重现计算模型。

Cartesi 本文介绍了Cartesi，用于开发和部署可扩展DApps（分布式应用）的第2层平台。 Cartesi DApps是一种混合模式，包括链上和链下两个部分。

链下模块在Cartesi节点网络（第6节）中运行，每个节点代表DApp用户的权益。链下部分可再细分为两个模块。 本地计算直接在主机硬件中运行。尽管本地计算可以访问节点的全部处理能力（包括GPU），但计算不可重现，至少不是pri-ori。可重现计算运行在Cartesi Machine中，受Cartesi 节点控制。这是一种完整的，运行在确定性的RISC-V平台上(第3节)的自包含的Linux系统（self-contained deterministic linux system），节点通过一些确定的主机接口和Cartesi Machine进行交互。

在区块链中，Cartesi DAppDApp开发者可以指定链下计算采用可重现方式（第5节），Cartesi 节点会自动根据所指定方式执行链下计算。DApp开发者可以请求节点提交结果，验证交易和辩论其他节点提交的结果。从链的角度来说，处理有争议的计算只需占用微不足道的资源。 当争议发生，争议处理成本只是存储和时间的对数复杂度，即O(logN)，离线部分的Cartesi 节点计算复杂度，也只是线性开销O(n)，且常数不超过2。

把计算移到链下会获得除了伸缩性之外的另外的好处。 Cartesi Machine让开发者使用其所熟练的开发语言，工具，库，软件和服务变得可能了。此外，由于Cartesi就其本质来说，计算的组织形式和底层区块链类型并无关连，那么通过把现有的复杂的合约逻辑隔离到链下进行可重现计算，开发者甚至可以使其DApp能够做到跨链交互。

本文件的重点是Cartesi的核心。 它包括Cartesi的完整规范，用于控制它的主机接口，用于指定复杂的链下计算的区块链接口，以及用于执行和验证这些计算的Cartesi 节点接口。在此核心功能之上构建的高级工具，接口和各种用例将在之后的文档[Teixeira和Nehab 2019a]中描述。Cartesi SDK [Teixeira和Nehab 2019b]将提供关于所有接口的详细文档，以及Cartesi节点和Cartesi的开发环境。

3 Cartesi Machine规格

Cartesi Machine是一个独立的，确定性的计算模型，可以托管现代操作系统。发生在操作系统内部有充分的理由的真实世界的计算。 开发者接受过使用工具链的培训，这些工具链可以在任何给定的工作中以尽可能高的抽象级别运行。 这些工具链将它们与不相关的硬件细节隔离开来，甚至与给定操作系统的细节隔离开来。 因此，发明一种特殊的新架构需要移植工具链和操作系统。 相反，Cartesi Machines基于经过验证的架构，其标准工具链和操作系统已经是可用的。

另一方面，Cartesi Machines执行的链下计算必须通过区块链进行验证。 因此，区块链必须承载整个架构的参考和实施。 如果它永远值得信任，那么这种实施必须易于审计。为此，架构和实施都必须是开放的并且相对简单。 这些要求共同指向RISC-V。 RISC-V ISA基于最小的32位整数指令集，可以添加几个扩展[Waterman和Asanovic'2017a]。 正交地，操作数和地址空间宽度可以扩展到64位（甚至128位）。 此外，该标准还定义了一种特权架构[Waterman和Asanovic'2017b]，它具有现代操作系统常用的功能，例如基于分页的多个权限级别。

表1：扩展指令计数。 以x + y形式的条目，请参阅同一设施的32位和64位变体。

Integer	Mul/Div	Atomics	Privileged	Total
47+12	8+5	11+11	5	71+28=99

表2：处理器状态。 内存映射到最低512物理内存中的字节，用于外部只读访问。

Offset	State	Offset	State
0x000	x0	0x160	misa
0x008	x1	0x168	mie
...	...	0x170	mip
0x0f8	x31	0x178	medeleg
0x100	pc	0x180	mideleg
0x108	mvendorid	0x188	mcounteren
0x110	marchid	0x190	stvec
0x118	mimplid	0x198	sscratch
0x120	mcycle	0x1a0	sepc
0x128	minstret	0x1a8	scause
0x130	mstatus	0x1b0	stval
0x138	mtvec	0x1b8	satp
0x140	mscratch	0x1c0	scounteren
0x148	mepc	0x1c8	ilrsc†
0x150	mcause	0x1d0	iflags†
0x158	mtval		

†Cartesi-specific state.

虚拟内存，定时器，中断，异常和陷阱等。自由选择更适合其需求的扩展组合得以实现。

RISC-VRISC-V于2010年始于加州大学伯克利分校，并于2015年成立基金会。包括谷歌，三星和特斯拉在内的大型企业最近也开始使用该项技术[Tilley 2018]。 该平台由高活跃度的社区开发者提供支持，他们耗费大量精力搭建了软件的基础底层，最着名的是Linux操作系统的端口和GNU工具链[RISC-V 2018d]。但关键的是，RISC-V并不是耍花枪的技术架构。它已在本地硬件之上运行，而且SiFive公司目前已将其商业化。这意味着，未来Cartesi将不仅限于仿真或二进制链下翻译。

Cartesi Machine可以分为处理器和"主板B"，处理器执行计算，执行传统的获取 - 执行循环同时，保留了各种寄存器。该"主板B"通过各种存储器（ROM，RAM，闪存）和设备定义周围环境。 为了使验证成为可能，Cartesi Machines以明确定义的方式将其整个状态映射到物理内存。这包括处理器，主板和所有连接设备的内部状态。幸运的是，此修改不会以任何重要方式限制操作系统或其承载的应用程序。

3.1处理器

遵循RISC-V术语，Cartesi Machines实施RV64IMASU ISA。 RV之后的字母指定扩展集。 此选择对应于64位计算机，具有乘法和除法的整数算术，原子操作以及可选的Supervisor和用户权限级别。 此外，Cartesi Machines支持Sv48地址转换和内存模式。

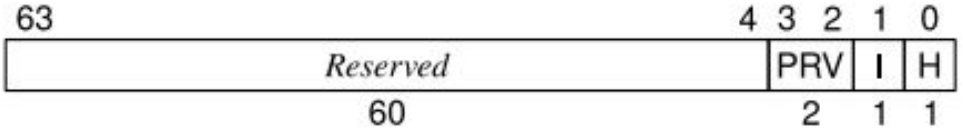

图1：iflags寄存器提供当前权限级别，并指定机器是暂时空闲等待中断还是已永久停止。

保护。 Sv48提供48位受保护的虚拟地址空间，分为4KiB页面，由四级页表组成。 这组功能在区块链实施所要求的简单性和链下计算所期望的灵活性之间创造了平衡的折衷。

总共有99条指令，其中28条指令分别缩小或加宽，分别为64位或32位指令。 表1分解了每个扩展的指令计数。 这是一个RISC ISA，大多数指令非常简单，可以在几行高级代码中进行模拟。实际上，唯一复杂的操作是虚拟到物理地址转换。由于格式数量减少（仅为4，占用32位），指令解码变得特别简单。

整个处理器状态适合512字节，分为n至64个寄存器，每个寄存器保持64位。这些寄存器中的大多数由RISC-V ISA定义，由32个通用nteger寄存器和26个控制和状态寄存器组成。 剩下的是Cartesi特有的。 处理器通过将各个寄存器映射到物理存储器中的最低512字节，使其整个状态可用，外部和只读。 相邻的1.5KiB保留供将来使用。整个映射在表2中给出。

名称以i开头的寄存器是特定于Cartesi的，并具有以下语义。 寄存器iflags的布局如图1所示.PRV给出当前的权限级别，当处理器空闲时（即，等待中断），I设置为1，H设置为1，表示处理器已永久存在暂停。 寄存器ilrsc保存LR / SC原子存储器操作的保留地址

默认初始化使用以下值填充状态：

•iflags中的PRV设置为3（对于机器权限级别）；

•misa设置为RV64IMASU;

•mstatus中的SXL和UXL设置为2（64位）；

•pc从0x1000开始（指向ROM）；

•marchid设置为ASCII格式的cartesi␣。

mvendorid用于测试匹配的链上和链外实施。 每次更新匹配对时，mimplid都会递增。剩余的默认状态设置为零。

3.2主板

主板和处理器之间的交互通过映射到处理器物理地址空间的设备进行。 表3显示了这种映射。 从地址0x1000开始有64KiB的ROM，执行开始。 这个ROM的核心作用是持有描述系统硬件的设备[DTSpec 2017]。 此外，ROM-base的引导程序将x10寄存器设置为0（mhartid的值），x11指向设备，然后跳转到RAM-base为0x80000000。 这是启动映像的入口点所在的位置。 最后，可以为闪存设备留出一些额外的物理内存范围。 这些通常会预先加载文件系统映像。

2 x86 ISA定义了至少2000条（可能很复杂的）指令

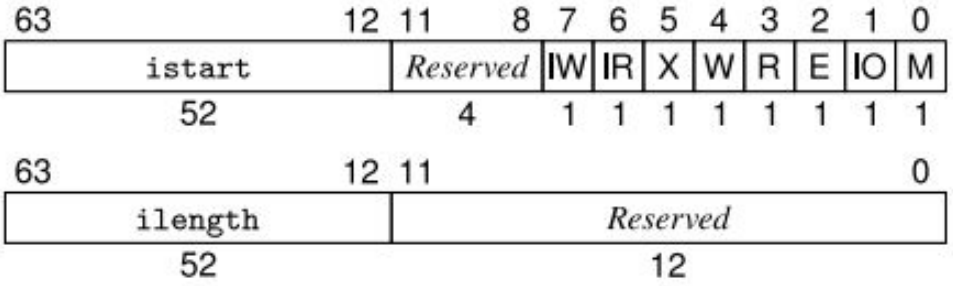

图2：物理内存属性。每个范围的istart和ilength与4KiB边界对齐。 每个64位字的12个LSB给出了该范围的属性。

两个非内存设备映射到地址空间。 核心本地中断器（或CLINT）控制定时器中断。有效地址分别为0x0200bff8和0x02004000，映射到寄存器mtime和mtimecmp。 只要mtime等于mtimecmp，CLINT就会发出硬件中断。 为了确保可重复性，处理器的时钟和定时器

由100的恒定除数锁定。换句话说，mtime每增加100个mcycle，就会增加一次。主机 - 目标接口（HTIF）调解与外部世界的通信。其活动地址为0x40000000（tohost）和0x40000008（fromhost）。它在写入tohost时停止机器，位63-48设置为0，位0设置为1.（位47-1可以设置为任意退出代码。）它也可以作为交互式部分的基本通信端口。

物理内存映射由物理内存属性记录（PMA）描述。每个PMA由2个64位字组成。 第一个单词给出一个范围的开头，第二个单词给出它的长度。 由于范围必须与4KiB页边界对齐，因此每个字的最低12位可用于属性。 图2显示了每个属性字段的含义。 M，IO和E位是互斥的，分别将范围标记为内存，I / O映射或排除。 R，W和X位分别授予读，写和执行权限。 最后，IR和IW位分别将读取和写入的范围标记为幂等。

该主板支持总共32个PMA，并使它们以只读方式可用，从物理内存中的偏移量2KiB开始。 另外2KiB留作将来使用。 PMA 0描述RAM，PMA 16-23描述闪存设备0-7。 这些PMA在初始化期间是用户可配置的，之后是只读的。 （RAM istart字段被硬编码为0x80000000。）这些记录一起限制了计算期间可访问的最大存储量。

3.3状态转换函数

机器计算的序列是$s_0, s_1, \ldots, s_h$，由过渡函数管理，使得

$$s_{i+1} = \text{step}(s_i). \qquad (1)$$

在这里，s_0是初始状态，s_h是停止状态。 前面的部分详细描述了Cartesi machine的状态空间和过渡功能。

回想一下，此状态由Cartesi machine的64位地址空间中每个字的值组成。 实际上，表示一个状态需要的花费少于2^{64}个字节。 只有表3中描述的区域必须明确定义。 所有剩余值都可以隐含地用零填充。

RISC-V ISA手册[Waterman和and Asanovic'2017a，b]指定了与每个结构执行相对应的状态转换。 这意味着在执行的指令之间很好地定义了状态。 由于所有指令都可以在O（1）时间内实现，因此Cartesi将每个状态转换定义为恰好1个周期。 可以从相应的序列中读取序列中给定状态的索引。

表3：Cartesi machine的物理内存布局。

Physical address	Mapping
0x00000000–0x000003ff	Processor shadow
0x00000800–0x00000Bff	Board shadow
0x00001000–0x00010fff	ROM (Bootstrap & Devicetree)
0x02000000–0x020bffff	Core Local Interruptor
0x40000000–0x40007fff	Host-Target Interface
0x80000000–*	RAM
–	Flash 0 (Disk 0)
...	...
–	Flash 7 (Disk 7)

mcycle的价值。 （请注意，由于机器偶尔会空闲，因此minstret不会跟踪mcycle。）

唯一显著的是Cartesi特定修改涉及停止机器。当iflags中的字段H设置为1时，不允许进一步的状态转换。 当指示HTIF停止机器时，将显式设置条件。

3.4 Linux端口

从头开始设置Linux系统涉及到的各个步骤。与独立系统不同，嵌入式系统通常不是自托管。相反，组件构建在单独的主机系统中，在该系统上安装了目标体系结构的交叉编译工具链。 关键组件是GNU编译器集合和GNU C库。 此基础结构可在RISC-V GNU工具链存储库[RISC-V 2018a]中找到。第一步是要创建这种基础项设置。

然后可以使用工具链交叉编译Linux内核。 内核源代码可以在RISC-V Linux存储库[RISC V 2018b]中找到。内核以管理模式运行，位于在机器模式填充程序提供的Supervisor二进制接口（SBI）：Berkeley引导加载程序（BBL）。 BBL可以在RISC-V代理内核库[RISC-V 2018e]中找到。由此产生的引导映像被预加载到RAM中。 SBI提供了一个简单的接口，内核通过该接口与CLINT和HTIF协同工作。除了实现SBI之外，BBL还安装了捕获无效指令异常的陷阱。此机制可用于模拟浮点指令（请参阅第4.3节）。 安装陷阱后，BBL切换到管理员模式并将控制权交给内核入口点。

最后一步是创建根文件系统。 此过程从主机系统中的根目录开始，该目录包含一些子目录（sbin，lib，var等）和文本文件（sbin / init，etc / fstab，etc / passwd等）。许多常见的UNIX实用程序（ls，cd，rm等）的微小版本可以组合成单个二进制文件[Vlasenko 2018]。 目标可执行文件通常依赖于工具链（lib / libm.so，lib / ld.so和lib / libc.so）提供的共享库。当然，必须将这些库复制到根文件系统。一旦根目录准备继续，就将其复制到实际的文件系统映像中（例如，使用gene2fs）。

这些步骤可以自动化。Cartesi的SDK以便捷的Docker容器形式为开发者提供了预配置的主机环境。复杂的Linux系统可以在Sifive的Buildroot [Petazzoni 2018]的分支，或Yocto项目的RISC-V端口[RISC-V 2018c]的帮助下构建。容器中的环境使开发者能够根据应用程序的需要自定义启动映像和根文件系统。成千上万的软件包可供安装。

```
memory@80000000 {
  device_type = "memory";
  reg = <0x0 0x80000000 0x0 0x8000000>;
};

flash@8000000000 {
  #address-cells = <0x2>;
  #size-cells = <0x2>;
  compatible = "mtd-ram";
  bank-width = <0x4>;
  reg = <0x80 0x0 0x0 0x4000000>;
  fs0@0 {
    label = "root";
    reg = <0x0 0x0 0x0 0x4000000>;
  };
};

chosen {
  bootargs = "root=/dev/mtdblock0 rw";
};
```

图3：使用128MiB RAM和64MiB闪存设备进行简单设置的部分设备，作为根文件系统安装。

在完成自己的初始化后，内核最终将控制权交给/ sbin / init。 在Cartesi DApps中，这通常是一个shell脚本，它调用适当的命令序列来执行以形成所需的计算。内核在bootargs中的分隔符␣-after之后将所有参数作为命令行参数传递给/ sbin / init。这些参数可用于为要执行的计算定义附加参数。完成后，/ sbin / init使用HTIF通过可选的退出代码暂停机器。 这可以用作计算输出的一部分。 任意复杂的输入，参数和输出都可以作为闪存设备传递。

cartesi机器的非链实现有两个目的，它们的主要作用是执行计算本身。第二个作用是支持解决有关计算结果的争议。为了提供这些服务，cartesi机器的非链实现必须公开可编程接口。

5 区块链中的Cartesi Machine

回想一下，Cartesi是一个开发分散应用程序的平台。Cartesi DApps使不相互信任的各方能够在区块链中签订一份取决于链下计算结果的约束性合同。使用”Alice”和”Bob”这些角色代表这些派对很方便。请注意，”Alice”和”Bob”是角色，而不是人。它们甚至可能代表

相互竞争的集体利益。实际上，这两个角色都将由Cartesi节点自动播放，以捍卫控制节点运行的链下计算机的人的利益。因此，Cartesi DApps是在区块链中运行的一组智能合约与在"Alice"和"Bob"的节点上运行的链下软件之间的协作。作为一般规则，同一DApp开发者负责智能合约和DApp特定的链下软件。查理将扮演DApp开发者的角色。"Alice"和""Bob""信任查理，否则他们不会与他的DApp交往。然而，查理既不信任"Alice"也不信任""Bob""。当然，"Alice"和""Bob""也不相互信任。

Cartesi的角色是支持查理的工作。为此，Cartesi提供了各种原语，Charlie用它来调解"Alice"和"Bob"之间潜在的对抗性交互。一些原语不需要交互，可以在区块链中从输入中自主地进行评估。然而，有趣的原语是那些虽然完全由它们的输入定义但只能在链外进行评估的原语。通过构造，当使用Cartesi DApp时，"Alice"和"Bob"总是同意这些原语的输入。在不失一般性的情况下，"Bob"评估原始的离线链并提交结果。然后"Alice"有机会接受或拒绝""Bob""的结果。查理的DApp可以使用无可争议的结果来达到他选择的目的。如果被拒绝，Cartesi将与"Alice"和"Bob"一起参与争议解决协议，该协议以正当理由对仲裁方进行仲裁。这种判断总是在几次交互中完成，并且对区块链的计算成本可以忽略不计。Cartesi以对查理极为方便的方式自动化大部分过程。

这些原语中最重要的是Cartesi机器。智能合约无法在区块链中存储Cartesi机器的状态，更不用说执行隐含的计算了。毕竟，处理能力和存储容量方面的成本都是令人望而却步的。为了解决这些问题，Cartesi使用加密哈希来简洁地表示区块链中的机器状态。从区块链的角度来看，计算只是一对与机器的初始和最终状态相对应的散列。由这种散列所对应的存储器的内容仅在链外已知。 Cartesi定义了各种附加原语，允许智能契约方便地操纵与这些哈希相对应的状态的内容。

5.1 哈希的机器状态表示

Merkle树[Merkle 1979]是二叉树，每个节点都包含一个哈希值。 在Cartesi中，Merkle树基于keccak哈希函数[Dworkin 2015]*（这简化了与以太坊区块链的整合。 与以太坊一样，Cartesi假设没有实用的方法来设计keccak哈希函数的冲突。）* 。让我们成为Cartesi机器状态，给出其64位地址空间的全部内容。用于s的Merkle树m，或者,等效地，它的根节点是

$$m = \mathsf{merkle}(s). \tag{2}$$

这棵树是用树叶建造的，方式如下。 首先，状态被划分为2^{61}个64位字。 树叶包含这些单词的哈希值。 由于不存在歧义，我们可以通过使用关联的散列标识树中的每个节点来简化表示法。 然后，树中的内部节点v由它们的两个子u_1和u_2通过关系构建

$$v = \mathsf{keccak}(u_1, u_2). \tag{3}$$

这里，keccak计算两个输入散列的串联的散列。 该过程构建深度为61的树。深度为d的节点集将状态划分为2^d范围，每个范围具有2^{64-d}字节。 因此，每个节点可以通过其深度和其范围的起始地址a来识别，该范围与2^{64-d}边界对齐。

图5显示了一个示例机器状态。 它具有特殊属性，即设备已与m中的节点对齐。 例如，节点v4 将机器中的所有内容与闪存设备分开。 它涵盖了处理器和主板，ROM，CLINT和HTIF设备以及RAM的阴影。 节点v0–v3和v5 均覆盖独立的闪存设备。

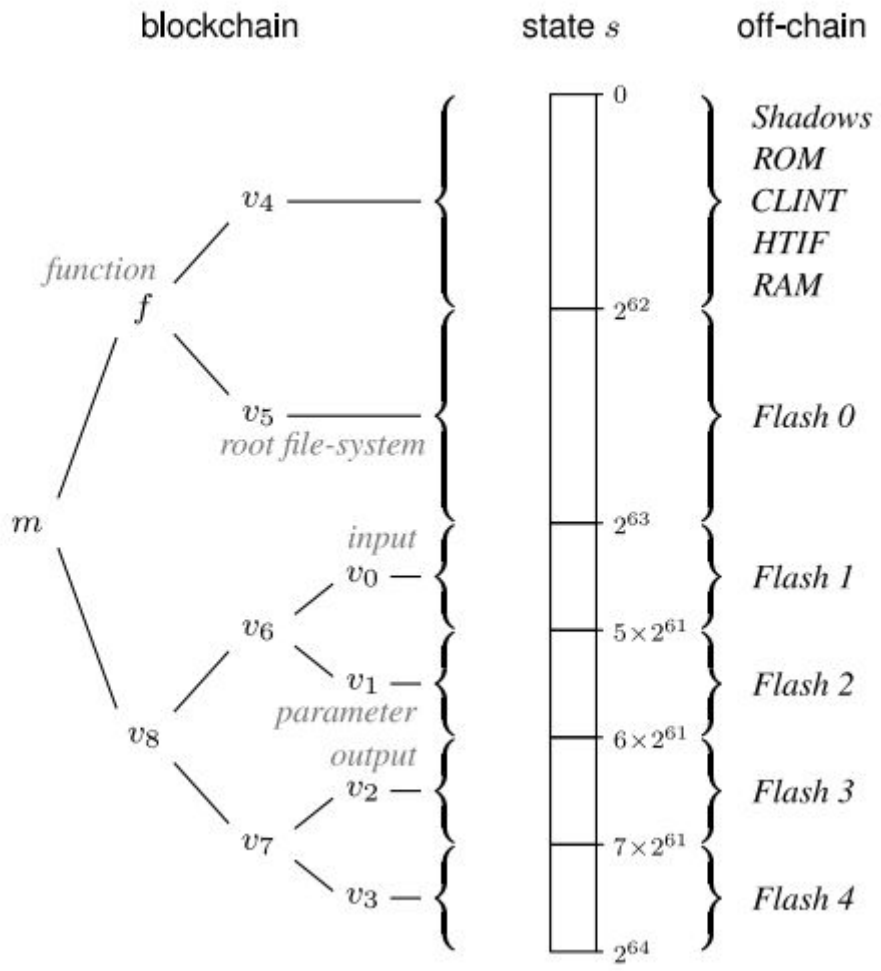

图5：机器状态的区块链和离线表示相互关联的方式。

将s视为某些机器的初始状态。在这种情况下，ROM将包含描述硬件的设备，并且RAM将预先加载Linux内核。根文件系统（flash 0）中/ sbin / init中列出的特定命令选项决定了机器的功能。例如，它可以在输入文件系统（flash 1）上执行任意计算，并将结果存储在输出文件系统（flash 3）中。甚至可以通过附加的独立参数文件系统（flash 2）的内容来通知计算。

现在假设在状态s'停止。从区块链的角度来看，运行m直到它停止可以看作是对任意函数v2′ = f (v0 , v1)的评估。这里，v2′ 是m'= merkle（s'）中的节点，其对应于m中的v2。考虑一个散列f库，每个散列都对应一个不同的有用函数。例如，一个这样的函数可以将输入文件系统v0 解密为输出文件系统v2′，从参数文件系统v1获取密钥。要根据具有单个散列m的Cartesi机器指定此计算，智能合约必须从其组件f，v0和v1构建m。一旦收到m'，就必须能够解决m是否确实停止为m'的争议。最后，它必须能够验证如果v1′是m'中对应于m中的v1的节点。

Merkle树的以下属性是所有这些操作的基础：给定节点v，其在树m中的深度d，以及相关存储器范围的起始地址a，可以验证v确实是m的一部分。要看到这一点，请考虑从v到m的路径

$$w_d = v,\ w_{d-1},\ \ldots,\ w_1,\ w_0 = m. \tag{4}$$

然后，给出路径中每个节点wi 的兄弟姐妹ui ：

$$w_{i-1} = \begin{cases} \text{keccak}(u_i, w_i), & \text{if } a \wedge 2^{64-i}, \\ \text{keccak}(w_i, u_i), & \text{if } \neg(a \wedge 2^{64-i}). \end{cases} \tag{5}$$

出于这个原因，序列

$$\text{siblings}(m, a, d) = (u_1, u_2, \ldots, u_d) \tag{6}$$

作为v在地址a和深度d在m的声明的证据。要验证这一点，只需从（5）计算p0并与m进行比较。

此外，给定v在m的有效证明，可以使用相同的过程来证明根哈希m'是通过用任何给定节点v'替换m中的v而得到的。这些验证非常有效，每个验证只需要keccak哈希的d应用程序。

我们现在准备定义前两个Cartesi原语

$$v = \text{slice}(m, a, d) \quad \text{and} \quad m' = \text{splice}(m, a, d, v'). \tag{7}$$

在没有争议的情况下，切片返回v并且splice返回用v'替换m中的v的结果m'。为了成功地在争议中捍卫这些结果，可以简单地将兄弟姐妹（m，a，d）和v呈现给区块链。

为方便起见，Cartesi定义了两个额外的基元

$$w = \text{read}(m, a) \quad \Leftrightarrow \quad \text{keccak}(w) = \text{slice}(m, a, 61), \quad \text{and} \tag{8}$$

$$m' = \text{write}(m, a, w') = \text{splice}\big(m, a, 61, \text{keccak}(w')\big) \tag{9}$$

直接操纵单词而不是哈希。一旦区块链收到siblings（m，a，61）和w，就可以解决争议。

5.2 验证游戏

验证游戏[Feige和Kilian 1997]是一种允许具有有限计算资源的仲裁者在两个计算上无限的玩家之间裁判游戏的协议。 Canetti等人介绍了它与Merkle树的结合使用。 [2011]， 区块链的申请首次出现在TrueBit [Teutsch和Reitwießner2017]中。 在这种情况下，区块链是“参考”，而“游戏”是在一个“玩家”“Bob””之间，一个是保护结果进行链下计算，另一个是“玩家”“Alice”。

设s0为计算的初始状态，s_n 为最终状态。 回想一下$s_{i+1} = \text{step}(s_i)$ 和 $m_i = \text{merkle}(s_i)$。 验证游戏是Cartesi原始的争议解决机制

$$m_n = \text{compute}(m_0, n) = \text{merkle}\left(\text{step}^{(n)}(s_0)\right). \qquad (10)$$

它分为两个阶段。第一阶段找到"Alice"和"Bob"不同意的单一计算步骤。最后阶段有效地计算了下面的步骤。如果它符合""Bob""提出的状态，他就会赢得争议。否则，"Alice"获胜。

不一致步骤 如果满足以下两个条件，则i <j的区间[i，j]被称为不一致区间：

1.""Bob""已向区块链发送哈希m_i和m_j，声称它们对应于$\text{merkle}(s_i)$和$\text{merkle}(s_j)$;

2. "Alice"在区块链中表明她同意m_i但不同意m_j。

不一致步骤是不一致的间隔，其中j－i = 1。

当"Alice"对$m_n = \text{compute}(m_0, n)$进行争议时，范围[1，n]成为初始不一致区间。分区合同可以从[1，n]开始，通过交互式二进制搜索找到不一致的步骤。在迭代l，合同以不一致的间隔$[i_l, j_l]$开始。它从"Bob"请求散列$m_{k_l} = \text{merkle}(s_{k_l})$，其中kl是$i_l$和$j_l$之间的中间点。知道"Bob"的mkl 后，"Alice"然后在$[i_{l+1}, j_{l+1}] = [i_l, k_l]$或$[k_l + 1, j_l]$之间选择下一个不一致的间隔。这一直持续到"Alice"选择长度为1的间隔。

该过程在"Alice"，"Bob"和分区契约之间的O（log n）交互之后完成*（这可以通过改为n-ary搜索来减少）*。[4] 在预定截止日期内未能做出反应的任何一方在超时时失去验证游戏。在迭代l中，"Alice"和"Bob"必须独立地获得机器的状态s_{k_l}。如果机器始终从头开始，则所发生的总链外计算为O(n log n)。但是，通过保留sil的快照，它们可以将成本降低到O（n）。

解决纠纷 此时，区块链已找到不一致的步骤[i，i + 1]，其中i∈{0，...，n－1}。根据""Bob""的说法，它知道mi 和mi+1 。"Alice"同意mi，但是对mi+1 进行了解释。为了确定正当派对，区块链必须有效地计算$m_{i+1} = \text{merkle step}(s_i)$并将其与"Bob"的$m_{i+1}$进行比较。但是，区块链没有对$s_i$的无限制访问权限。它只有根哈希mi = merkle(si)。

期望"Alice"将她的链下状态访问日志发布到内存管理器合同中（步骤(s_i)）。在链外，这些访问逐渐将状态$s_i = (s_i)0$修改为$(s_i)1, (s_i)2, \ldots$。直到第k次和最后一次访问之后，它变为$(s_i)_k = s_{i+1}$。所有机器步骤都需要O（1）时间来模拟，并且访问次数k总是很小。对于j∈{1，...，k}，日志中的条目j包含

1. 访问（读或写）的操作o_j

2. 访问的地址a_j；

3. 在$(s_i)_{j-1}$中a_j的单词rj；

3'. 在写入的情况下，在$(s_i)_j$中的aj 处的单词w_j；

4. 兄弟姐妹$(m_i)_{j-1}, a_j, 61$.

步骤函数的区块链实现由仿真器契约托管。"Alice"的链下实现必须将此区块链实现与记录状态访问的顺序相匹配。作为此限制的一个好处，区块链实现可以读取和写入状态，就像它的整个内容可用一样。在执行期间，引用步骤函数向存储器管理器发出一系列状态访问。只要访问与"Alice"的日志匹配，一切都会透明地工作。

形式上，当参考步骤函数执行k'访问时，存储器管理器逐渐将$m_i = (m'_i)_0$更新为$(m'_i)_1$，$(m'_i)_2$，..，直到它达到$(m'_i)_{k'}$。访问j，对于j∈{1，...，k'},

包含以下信息

1. 访问（读或写）的操作o'_j；
2. 访问地址a'_j；
3. 在写入的情况下，要写入单词w'_j.

在处理每个访问时，内存管理器：
•检查 j ≤ k，$o'_j = o_j$ 和$a'_j = a_j$；

•检查 $r_j = \text{read}(m'_{j-1}, a_j$ 与兄弟姐妹。

然后，对于读访问，内存管理器：
•设置$(m'_i)_j = (m_i)_{j-1}$；

•将r_j 返回给模拟器合约。

对于写入，内存管理器：
•检查$w' = w_j$

•与兄弟姐妹一起设置$(m'_i)_j = \text{write}(m'_{i})_{j-1}, a_j, w_j)$

在任何时候，如果检查失败，"Alice"就会失去争议。但是，如果
k = k'和 $m_{i+1} = (m'_i)_{k'}$，"Alice"赢得争议。

5.3 Cartesi Machine作为众多基元之一

Cartesi原语通过函数编程接口提供给Charlie。目标是将原语与特定智能合约编程语言和区块链的特性隔离开来。语法本身并不重要。重要的是与每个原语相关的语义。

Charlie可以使用此接口来构建表示复杂复合计算的表达式DAG。计算可涉及多个彼此交换数据的机器。首先通过调用以下内容初始化空DAG：

```
dag = dag()
```

每个DAG顶点对应一个基元。基元的输入来自其子顶点的输出。基元的输出又可以作为一个或多个基元的输入。

基元分为两类。有争议的原语表现为"Bob"向"Alice"承诺的未来价值。它们的输出由"Bob"设置，并且必须由"Alice"接受或提出异议。可争议的图元只能在表达式DAG中显示为内部顶点。它们是5.1和5.2节中描述的读，写，切片，拼接，步和计算基元。

常数基元的输出由查理设定。"Alice"和"Bob"在与查理的DApp的互动中都隐含地接受了它们。这些原语可能代表大块数据，Charlie可能会保证其内容的可用性，可能需要我们

的数据可用性原语的帮助。当然，word，hash，string和id文字是常量。 Cartesi的blob，资源和机器原语（如下所述）也是不变的。 DAG离开时只能显示常量基元。

Cartesi支持常量和未来类型：

```
constant ::= word | hash | string | id
disputable ::= word-future | hash-future
```

常量基元仅接受常量作为输入。 相反，有争议的原语同时接受常量和争议：

```
word-type ::= word | word-future
hash-type ::= hash | hash-future
```

常量基元Blob基元表示存储在区块链中的任意二进制数据。 提供的哈希给出了输出。它必须匹配从数据构建的Merkle树的根哈希，用零填充到$2^{64-depth}$ 字节：

```
hash = dag:resource{
  hash = hash,
  depth = word,
  range-length = word,
  download-size = word,
  uri = string
}
```

资源原语描述了离线存储的文件。 该规范遵循4.1节中描述的脚本接口。 唯一的区别是后备文件是作为路径给出的。 相反，它们是资源常量。 提供的散列输出必须对应于相应Cartesi Machine状态的根Merkle树散列：

```
hash = dag:machine{
  hash = hash,
  processor = processor,
  rom = rom,
  ram = ram,
  flash0 = drive,
  ...
  flash7 = drive,
  clint = clint,
  htif = htif
}
```

有争议的原语 计算原语执行Cartesi Machine。 初始状态给出m0 的值并且步长n的值，以便将来的值是compute(m0 , n) 。 因此，步骤必须先验地计算所需的计算量：

```
hash-future = dag:compute{
  initial-state = hash-type,
  steps = word-type
}
```

Merkle树操作原语切片，拼接，读取和写入可用作：

```
hash-future = dag:slice{
  root = hash-type,
  address = word-type,
  depth = word-type
}

hash-future = dag:splice{
  root = hash-type,
  address = word-type,
  depth = word-type,
  target = hash-type
}

word-future = dag:read{
  root = hash-type,
  address = word-type
}

hash-future = dag:write{
  root = hash-type,
  address = word-type,
  word = word-type
}
```

Cartesi还提供了各种简单的原语，增加了表达的表达能力。可以在区块链中直接从其输入中解决对这些原语的争议。对单词的几个二元操作具有签名：

```
word-future = dag:bin-op(word-type, word-type)
```

并镜像RISC-V ISA。它们可以分为算术：

```
add, sub, mul, mulh, mulhu, mulhsu,
div, divu, rem, remu, sll, srl, sra;
```

Disclaimer: For accuracy, the original English version of Cartesi Core's Techpaper must be referred to.

按位：

```
or, and, xor;
```

和比较：

```
eq, ne, lt, ltu, ge, geu.
```

有符号整数用二进制补码表示。布尔值作为单词返回，其中1表示true，0表示false。相反，当条件期望布尔值时，0被视为假，而任何其他值被视为真。

如果条件为真，则输出设置为if-true的基元可以作为三元条件，如果是，则为if-false：

```
word-future = dag:if{
  condition = word-type,
  if-true = word-type,
  if-false = word-type
}
```

散列基元通过其64位组件字的串联构建256位散列：

```
hash-future = dag:word4(word-type, word-type,
    word-type, word-type)
```

为了帮助Merkle树构造，可以从单词和两个哈希的串联构建哈希：

```
hash-future = dag:keccak(word-type)
hash-future = dag:keccak(hash-type, hash-type)
```

为了完整性，还可以测试哈希的相等性并将其用作条件的输入：

```
word-future = dag:eq(hash-type, hash-type)
word-future = dag:neq(hash-type, hash-type)

hash-future = dag:if{
  condition = word-type,
  if-true = hash-type,
  if-false = hash-type
}
```

DAG和顶点接口 Charlie必须为"Alice"和"Bob"的角色定义玩家的身份。回想一下"Bob"提出了一个结果，"Alice"可以反对或接受它：

```
dag:proposing-role{id = id, stake = word}
dag:objecting-role{id = id, stake = word}
```

利益论证给出了购买"Alice"或"Bob"的位置的价格。它根据计算结果衡量每个人的利害关系。 Charlie设置此值以便"Alice"和"Bob"委派他们的角色（第6.2节）。

原始创建函数返回：

```
vertex ::= constant | disputable
```

DAG可以包含多个断开连接的子DAG。要指定或者检索DAG的根顶点，Charlie调用：

```
dag:root(vertex)
```

以后可以通过以下方式获取此值：

```
vertex = dag:root()
```

可以查询任何顶点的基元：

```
primitive = vertex:primitive()

primitive ::= word | hash | string |
  blob | resource | machine |
  read | write | slice | splice |
  add | sub | ... | geu | if |
  word4 | keccak | keccak-hh | eq-hh | if-hh
```

同样，孩子可以通过姓名或索引获得：

```
vertex = vertex:child-by-index(word)
vertex = vertex:child-by-name(string)
```

原始子函数和子函数一起使得能够遍历从顶点到达的整个子DAG。

DAG和顶点可以在其生命周期内改变状态：

```
dag-state ::= undefined | proposed | accepted |
  objected | sustained | overruled

vertex-state ::= undefined | proposed | accepted
```

常量顶点始终处于可接受状态。在构造时，DAG及其所有有争议的顶点都处于未定义状态。这些状态可以从DAG和顶点对象获得：

```
dag-state = dag:state()
vertex-state = vertex:state()
```

建议任何顶点的值将其状态更改为建议：

```
vertex:propose-hash(hash)
vertex:propose-word(word)
```

要开始提案，"Bob"首先调用：

```
dag:start-proposal()
```

然后，他提出了根顶点的输出值。最后，他通过调用以下内容完成提案：

```
dag:finish-proposal()
```

这会将DAG更改为建议的状态。

可以通过调用以下方式检查为任何顶点建议的值：

```
word = vertex:proposed-word()
hash = vertex:proposed-hash()
```

为了接受DAG的建议根，"Alice"调用：

```
dag:accept()
```

这会将DAG状态更改为已接受。

为了反对DAG的根值，"Alice"调用：

```
dag:start-objection()
```

然后为根顶点提出一个新的，不同的值。现在有两种情况需要考虑。如果根顶点的所有直接子节点都处于接受状态，则可以通过根原语解析协议来解决争议。为此，"Alice"只是打电话：

```
dag:finish-objection()
```

在协议完成时，这会将DAG更改为对象状态。如果"Alice"成功，则DAG将更改为持续状态。否则，它将更改为否决状态。

但是，如果存在任何建议或未定义的子节点，则必须首先为从根节点可到达的子DAG中的所有顶点建议值。只有这样她才能打电话：

```
dag:finish-objection()
```

这会将DAG更改为对象状态。"Bob"现在必须找到一个可从根访问的顶点，他接受所有输入但是输出的对象。为了指定这个顶点，"Bob"从根提供了它的路径。他还必须为其输出指定一个不同的值：

```
dag:defend-word(path, word)
dag:defend-hash(path, hash)

path ::= {vertex_1, vertex_2, ..., vertex_k}
```

如果顶点未定义，则"Alice"的争议被宣布为格式错误，"Bob"立即获胜。如果路径无效，则"Bob"的防御声明为格式错误，"Alice"立即获胜。否则，使用原始争议解决协议。

通过这种设置，任意复杂的表达式就像有争议的Cartesi原语一样：他们已经商定了输入，并为其输出配备了争议解决程序。

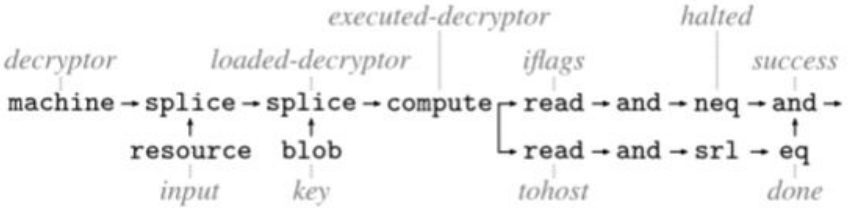

图6：对应于解密器示例的表达式DAG。为简洁起见，省略了与文字值对应的常量顶点。变量名称以灰色显示。

示例 以下示例说明了表达式DAG接口的强大功能：

```
d = dag()
decryptor = library{
hash = decryptor-machine-hash,
dag = d
}
input = d:resource{
hash = input-hash,
depth = decryptor-flash-1-depth,
uri = http://example.com/charlie/input.drive
}
key = d:blob{
hash = password-hash,
depth = decryptor-flash-2-depth,
data = password
}
loaded-decryptor = d:splice{
root = d:splice{
root = decryptor,
address = decryptor-flash-1-addr,
target = input
},
address = decryptor-flash-2-addr
target = key
}
executed-decryptor = d:compute{
initial-state = loaded-decryptor,
steps = 10*2^32
}
iflags = d:read{
root = executed-decryptor,
address = machine-processor-iflags-addr,
}
tohost = d:read{
root = executed-decryptor,
address = machine-htif-tohost-addr
}
halted = d:neq(d:and(iflags, 1), 0)
done = d:eq(d:srl(d:and(tohost, d:srl(-1, 16)), 1), 0)
success = d:and(halted, done)
d:root(success)
```

在该示例中，Charlie定义了具有根成功的表达式DAG。隐含计算以解密器机器开始，该解密器机器可从离线库（未示出）获得。 Charlie可以在Cartesi节点中安装自己的预定

义机器库，或者使用由另一个开发者远程创建和存储的机器规范库。无论哪种方式，哈希常量确保可以轻松识别已被篡改的机器。接下来，将输入和关键闪存设备拼接到解密器机器中。这个加载的解密器最多运行2 * 2 ^ 32步，并产生一个执行的解密器。然后探测相应的状态。检查处理器的I flags寄存器会判断机器是否已停止。存储在HTIF的to host寄存器的有效负载中的值告诉机器在解密后是否停止。图6显示了相应的DAG，省略了文字值（即字符串，单词和散列）。

Charlie监督"Alice"和"Bob"与DAG及其顶点的交互。毕竟，Charlie负责这些对象所在的区块链DApp组件。 Charlie代表"Alice"或"Bob"代表的软件可以发出触发来自代表"Alice"和"Bob"的Cartesi节点的反应的事件。这些反应也在Charlie的控制之下，因为他负责安装在他们节点中的链下DApp组件。像往常一样，他必须保护他的DApp免受流氓用户的攻击。 Cartesi通过在所有DAG操作中封装访问控制来简化此过程的一部分。

7 未来的工作

本文档的重点是核心功能，以及DApps用于直接指定，控制和验证链外计算的接口。Cartesi平台将提供在核心上构建的几个附加组件，或扩展其范围。这些将在未来的出版物[Teixeira和Nehab 2019a,b]中更详细地描述。

数据可用性 Cartesi通过保持仅链上Merkel树的离线数据哈希来弥补区块链的严重存储限制。如第5.2节所述，Cartesi假定参与验证角色的所有各方都可以访问这些数据。在某些应用中，这很难保证。特别是，必须减轻数据扣留攻击的风险，其中一方向区块链提交哈希值，同时拒绝向其他人提供此数据。

数据可用性问题是区块链共识算法设计中的一个主要问题[Buterin 2012]。但是，在地方共识的背景下，这个问题变得更加简单。 Teixeira和Nehab [2019a]提供了几种设计模式，用于在验证期间处理数据可用性。数据通道，设备加密和数据分类帐可确保Cartesi DApps可能遇到的所有情况下的可用性。

可用性 广泛采用块链技术的主要障碍之一是DApp用户遇到的不便。尽管关于集中式应用程序可用性的文献仍然适用于分散式应用程序，但从用户体验的角度来看，区块链特性还没有得到充分解决。 Teixeira和Nehab [2019a]描述了开发简单直观的DApps的几种设计模式。

例如，Cartesi将为交易代币提供自动化基础设施。这将使用户免于担心每个DApp内使用的不同令牌。还将提供外包延期行动的系统。这将使用户即使在参与需要在严格期限内与区块链交互的协议时也可以关闭他们的机器。在这种情况下，代理方将代表用户行事以换取费用。 （与第6.2节中描述的争议委托市场非常相似。）使用加密时间锁[Rivest等. 1996年]还将适应用户必须在未来揭示不应立即传递给代理方的秘密的情况。将描述其他可用性构造以促进文件传输并降低Gas成本。这些设施将使Cartesi DApps的用户体验更接近当前的集中式解决方案。

Cartesi SDK 随着Cartesi SDK的发布，可以使用各种更高级别的API来封装核心的典型用例。这些将包括上述可用性和数据可用性解决方案，以及Cartesi节点的容器和Cartesi Machine的开发。随着时间的推移，SDK中可用的API将大大减少DApps区块链组件的大小和复杂性。相反，这将显着增加DApps对多条区块链的跨链可移植性。Cartesi SDK将在开源中分发并广泛记录[Teixeira和Nehab 2019b]。

Cartesi Machine的扩展 Cartesi Machine可以通过两个令人兴奋的新设备进行扩展。Dehashing设备为应用程序提供了遍历散列指针数据结构的能力。在Cartesi Machine内运行的程序可以使用Dehashing设备来读取仅给定其散列的区块的内容。虽然这种操作一般是不可能的，但是当所有各方事先知道允许区块的范围时，它变得可能。最直接的应用是区块链本身。当Cartesi Machine运行时，Dehashing设备查询预加载在主机中的哈希表，查找与哈希匹配的区块。如果出现争议，任何一方都可以提出该区块作为与所需哈希匹配的证据。通过这种方式，dehashing设备可以实现区块链内省。缔约方可以签订合同，这

些合同取决于合同本身定义的区块链的整个状态。这就有很多有价值的应用，特别是在期货市场。

另一个计划的设备是及时的数据端口。该端口通过将进入或离开机器的数据包与事件中的mcycle值绑定，实现Cartesi Machine之间的可重复通信。 DApps可以安排在给定的未来mcycle发生数据包传送。 Cartesi Machine也可以回滚到mcycle进行交付。及时的数据端口在向Web 3.0的发展方面开辟了新天地。它将启用涉及多个Cartesi Machine之间直接协作的DApps。

群体争议 可以设想涉及许多独立参与者的应用程序，每个参与者在链外计算的结果中都有一些利益。 在这种情况下，至关重要的是要防止一群不诚实的参与者使用连续的争议而不是诚实的结果作为对合同的拒绝服务攻击。 我们开发了一种验证游戏的变体，使任何诚实的参与者能够以可忽略的成本为整个人群辩护他的结果。 当需求变得明显时，Cartesi平台将扩展为支持这种变体。

8 结论

本文为Cartesi平台奠定了理论基础。 Cartesi的使命是帮助DApp开发者为他们的用户构建更加有竞争力的产品。伴随任意范式的转变，区块链为实际创新和风险带来了机遇“车轮改造”。秉持着“简约而简单”的开发原则，Cartesi的核心目的是要使开发者更容易的上手，更简便的提升开发效率。未来的文档[Teixeira和Nehab 2019a]中描述的Cartesi平台的剩余组件将帮助开发者在利用区块链的独特潜力时释放他们的创造力。

Disclaimer: For accuracy, the original English version of Cartesi Core's Techpaper must be referred to.

参考

BELLARD，F。2016. Softfp库。网页。 https：//开头贝拉德。组织/ softfp /。
BELLARD，F。2017.Riscvemu。源代码。 https：//开头贝拉德。组织/ riscvemu /。
BELLARD，F。2018.一个通用的开源机器模拟器和虚拟器。网页。 https://www.qemu.org。
BEN-SASSON，E.，BENTOV，I.，HORESH，Y。和RIABZEV，M。2018.可扩展，透明和后量子安全计算完整性。 IACR Cryptology ePrint Archive，2018：46。
BEN-SASSON，E.，CHIESA，A.，TROMER，E。和VIRZA，M。2013.对于冯·诺曼建筑，简洁的非交互式零知识。 Cryptology ePrint Archive，Report 2013/879。 https://eprint.iacr.org/2013/879。
BITANSKY，N.，CANETTI，R.，CHIESA，A。和TROMER，E。2012.从可提取的碰撞阻力到简洁的非交互式知识论证，再回来。参加第三届理论计算机科学创新会议，ITCS '12，纽约，纽约，美国。ACM，326-349。ISBN 978-1-4503-1115-1。 http://doi.acm.org/10.1145/ 2090236.2090263。
BLUM，M.，FELDMAN，P。和MICALI，S。1988.非交互式零知识及其应用。在第二届ACM年度计算机理论研讨会论文集中，STOC '88，纽约，纽约，美国。 ACM，103-112。国际标准书号0-89791-264-0。 http://doi.acm.org/10.1145/62212.62222。
BRASSER，F.，MÜLLER，U.，DMITRIENKO，A.，KOSTIAINEN，K.，CAPKUN，S。和SADEGHI，A.-R。 2017.软件盛大曝光：SGX缓存攻击是实用的。在第11届USENIX进攻性技术研讨会上。
BUTERIN，V。2012.关于数据可用性和擦除编码的说明。 https://github.com/ethereum/research/wiki/A- note-on-data-availability-and-erasure-coding。
BUTERIN，V。2018.关于分割区块链。维基。 https：// github.com/ethereum/wiki/wiki/Sharding-FAQs。
BUTERIN，V。2018. Sharding。GitHub存储库。 https://github.com/ethereum/sharding.git。
CANETTI，R.，RIVA，B。和ROTHBLUM，G。N. 2011.使用多个服务器的实际计算委派。在ACM计算机和通信安全会议的会议记录中，445-454。
CARDANO的VM。 2017. IELE OpCodes。源代码。 https://github.com/runtimeverification/iele-semantics / blob / master / iele.md。
CHENG，R.，ZHANG，F.，KOS，J.，HE，W.，HYNES，N.，JOHNSON，NM，JUELS，A.，MILLER，A。和SONG，DX 2018. Eki- den：A保密，可信和高效的智能合约执行平台。 CoRR，abs / 1804.05141。
DTSPEC。 2017.Devicetree规范。 Power.org，飞思卡尔半导体，IBM，Linaro和ARM。 HTTP：//的DeviceTree。有机
DWORKIN，M。J. 2015. SHA-3标准：基于排列的散列和可扩展输出函数。技术报告。
ENIGMA，T。2018. Enigma文档。 https://enigma.co/ protocol /。
FEIGE，U。和KILIAN，J。1997年。制作游戏简短。在STOC的会议记录中，506-516。
GOLEM，T。2016.The golem项目。 https://golem.network/ crowdfunding / Golemwhitepaper.pdf。
HOUSER，J。2017.Berkeley softfloat。网页。 HTTP：// WWW。 jhauser.us/arithmetic/SoftFloat.html。发布3d。
IEEE，C。S. 2008.浮点运算标准，IEEE Std 754-2008。
IEXEC，T。2017.基于区块链的分散式云计算。 https://iex.ec/whitepaper/iExec-WPv3.0- English.pdf。
LARIMER，D。2017.Dpos一致性算法 - 缺少白皮书。 https://steemit.com/dpos/@dantheman/dpos- consensus-algorithm-this-missing-white-paper。
LEE，S.，SHIH，M.-W.，GERA，P.，KIM，T.，KIM，H。和PEINADO，M。2017.用分支遮蔽推断sgx包围区内的细粒度控制流。在第26届USENIX安全研讨会上，USENIX Security，16-18。
MERKLE，R。C. 1979.Secrecy，Authentication and Public Key Systems。博士论文，斯坦福大学。
MIERS，I.，GARMAN，C.，GREEN，M。和RUBIN，A。D. 2013. Zerocoin：来自比特币的匿名分发电子现金。 http：//zerocoin.org/media/pdf/ZerocoinOakland.pdf。
MOGHIMI，A.，IRAZOQUI，G。和EISENBARTH，T。2017.CacheZoom：SGX如何放大缓存攻击的威力。在密码硬件和嵌入式系统国际会议上，69-90。
MORRISON，D。R. 1968. Patricia-实用算法检索字母数字编码信息。 ACM期刊，15（4）：514-534。
NAKAMOTO，S。2009.比特币：点对点电子现金系统。白皮书。 http://bitcoin.org/bitcoin.pdf。
NEO'S VM。 2017. OpCodes。源代码。 https：//开头github上。 com / neo-project / neo-vm / blob / master / src / neo- vm / OpCode.cs。
PARNO，B.，HOWELL，J.，GENTRY，C。和RAYKOVA，M。2013. Pinocchio：几乎实用的可验证计算。 2013年IEEE安全与隐私研讨会，238-252。
PETAZZONI，T。2018. Buildroot。网站。 https：//开头buildroot的。有机
RISC-V。 2018. GNU工具链。 GitHub存储库。 https：// github.com/riscv/riscv-gnu-toolchain。
RISC-V。 2018. Linux。 GitHub存储库。 https://github.com/ riscv / riscv-linux。
RISC-V。 2018. Poky：Yocto项目的港口。 GitHub存储库。 https://github.com/riscv/riscv- poky。
RISC-V。 2018.项目主页。 GitHub存储库。 https：// github.com/riscv。
RISC-V。 2018.代理内核。 GitHub存储库。 https：// github.com/riscv/riscv-pk。
RIVEST，R。L.，SHAMIR，A。和WAGNER，D。A. 1996. Time-lock puzzles and timed-release crypto。
SCHAEFFER，T。2018. Zokrates。 https://github.com/ JacobEberhardt / ZoKrates。
SONG，C.，WU，S.，LIU，S.，FANG，R。和LI，Q.-L。 2018.在可信硬件中分布式云计算。 https：//开头ankr。网络/。
SONM，T。2018. Sonm文档。 HTTPS：//docs.sonm。 COM /。
TEEX，T。2018. TEEX-TEE支持的公共区块链执行平台。 https://teex.io。
TEIXEIRA，A。和NEHAB，D。2019. Cartesi设计模式。
白皮书。出现。
TEIXEIRA，A。和NEHAB，D。2019. Cartesi SDK。出现。
TEUTSCH，J。和REITWIESSNER，C。2017.区块链的可扩展验证解决方案。白皮书。 HTTPS：//people.cs。 uchicago.edu/~teutsch/papers/truebit.pdf。
TILLEY，A。2018.谷歌，特斯拉对Arm芯片设计的挑战落后。信息。 https：//www.theinformation.com/articles/google-tesla- get-behind-challenge-to-arm-chip-design。
TRÖGER，J.，MIHOC KA，D。和KEPPEL，D。2011.使用事务提交/中止的快速微码解释。第四届二元翻译建筑与微体系结构支持研讨会，加利福尼亚州圣何塞。 HTTP：//www.emulators。 COM /文档/ AMAS-bt2011.pdf。
VAN SABERHANGEN，N。2013.Cryptonote v 2.0。
VLASENKO，D。2018. Busybox。网站。 https://busybox.net。
WATERMAN，A。和ASANOVIC，K。2017. RISC-V指令集手册，第I卷：用户级ISA。 RISC-V基金会。版本2.2。
WATERMAN，A。和ASANOVIC，K。2017. RISC-V指令集手册，第二卷：特权架构。 RISC-V基金会。版本1.10。
WATERMAN，A。和LEE，Y。2011.Spike，RISC-V ISA仿真器。 GitHub存储库。 https://github.com/riscv/riscv- isa-sim。
WEBASSEMBLY，C。G. 2018. WebAssembly规范，版本1.0。 https://webassembly.github.io/spec/core/index。 HTML。
WEICHBRODT，N.，KURMUS，A.，PIETZUCH，P。和KAPITZA，R。2016.AsyncShock：利用英特尔SGX飞地中的同步错误。在欧洲计算机安全研究专题讨论会上，440-457。
WOOD，G。2018.以太坊：一种安全的分散式广义交易分类账。 Yellowpaper。 HTTPS：//ethereum.github。 IO / yellowpaper / paper.pdf。拜占庭版e94ebda - 2018-06-05。