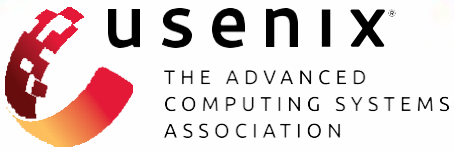


Aardvark：应用于基于账户的加密货币的异步认证字典

Derek Leung, *MIT CSAIL;* Yossi Gilad, *耶路撒冷希伯来大学;* Sergey Gorbunov, *滑铁卢大学;* Leonid Reyzin, *波士顿大学;* Nickolai Zeldovich, *MIT CSAIL*

https://www.usenix.org/conference/usenixsecurity22/presentation/leung

这篇论文被收录在《第31届USENIX安全研讨会论文集》中。

第31届USENIX安全研讨会论文集的开放访问由USENIX赞助。

Aardvark：应用于基于账户的加密货币的异步认证字典

Derek Leung
*MIT CSAIL**

Yossi Gilad
耶路撒冷希伯来大学

Sergey Gorbunov
滑铁卢大学

Leonid Reyzin
波士顿大学

尼古拉-泽尔多维奇
MIT CSAIL

摘要

我们设计了Aardvark，这是一种新型的验证字典，对查询和修改的正确性进行了简短的证明。我们的设计通过将验证者的数据外包给不受信任的服务器来减少加密货币交易验证的存储需求，这些服务器根据需要提供该数据的正确性证明。在这种情况下，短的证明特别重要，因为证明被分配给许多验证者，而长的证明的传输很容易支配成本。每当任何（甚至是不相关的）数据发生变化时，认证字典中的一块数据的证明都可能发生变化。这给加密货币交易的并发发行带来了问题，因为证明变得陈旧。为了解决这个问题、Aardvark采用了一个版本机制来安全地接受在有限的时间内，陈旧的证明。

在一个有1亿个键的字典上，操作证明的大小在Merkle Tree中约为1KB，而在Aardvark中为100-200B。我们的评估显示，一个32核的验证器可以处理1492个2941次/秒的操作，相对于维护整个状态，节省了约800倍的存储成本。

1 简介

确保大量记录集合的完整性是计算机安全的一个基本问题。1991年，Blum等人[10]描述了一个场景，即值得信任的计算机在不值得信任的存储中的数据上执行程序。为了防止储存器篡改数据，他们引入了一种数据结构，现在被称为双方*认证的字典*[24, 37]，它允许可信的计算机通过对数据存储一个小的*承诺来*确保数据的完整性。当计算机从数据中读取时，它要求不受信任的存储提供一个保证正确性的*证明*；这个证明是用承诺来验证的。在写入时，计算机也会收到一个证明，并使用该证明和当前的承诺，计算出对修改后数据的新承诺。

*作者在Algorand公司完成了这项工作的大部分内容。

已有人提出了认证数据结构，以减少*加密货币*的存储成本[12, 15, 33, 47, 50]。在加密货币中，任何用户都可以提交交易（例如向另一个用户发送钱）。交易被分配给众多验证者，他们运行一个容错协议来验证它们，根据系统的当前状态检查它们的有效性（例如，防止过度消费），并对它们进行审核。在典型的部署中，每个人都存储了系统的整个状态，以验证交易。

我们提出了Aardvark，一个用于加密货币的认证字典，它大大减少了验证器的存储（以实现所谓的*无状态验证*）。Aardvark支持许多验证器和每秒钟许多并发发行的交易。我们特别针对*基于账户的*加密货币（如Ethereum [51], Coda [13], Algorand [22]），它主要保留一个字典，将账户标识符（如公钥）映射到账户数据（如余额）。我们的设计解决了这个环境中存在的七个关键挑战。

首先，网络带宽是很重要的，因为许多验证者必须观察到交易和它们的证明。支持简短的证明是一个特别的挑战。在Aardvark、
与典型的基于Merkle-tree的认证数据结构相比，证明的时间大约缩短了10倍，因为其设计是基于
在基于配对的向量承诺上，有最短的已知证明。矢量承诺不提供字典界面（相反，它们提供认证的固定长度数组），所以我们的大部分工作涉及从这个更有限的功能中建立一个字典界面。

其次，不受信任的客户发布交易的能力要求数据结构无论如何使用都要保持高效。Aardvark对资源的使用实行严格的上限，即使在对抗性的访问模式下。

第三，多个客户可以同时发布交易，但验证者必须按顺序排列，这给证明的有效性带来了问题。例如，假设一个用户发布了一个修改记录的交易，然后另一个用户在验证者处理第一个交易之前发布了一个单独的交易。如果这两个交易都对同一记录进行操作，那么一旦第一个交易被

处理，第二个交易中存在的数据就会变得陈旧。此外，即使两个交易都访问不同的记录，在第一个交易被处理后，第二个交易中存在的证明可能会失效，因为一个证明可能取决于整个字典的状态。在交易排队时，重新计算或更新证明，因为它们已经过时了（例如，如[15]所建议的那样），可能会导致拥堵崩溃：当系统处于重载时，更多的交易将被排队，导致系统在更新这些排队交易的证明时进一步减慢。Aard- vark没有试图减少这些成本，而是通过*对*字典*进行版本管理来*完全避免这些成本。这种机制使系统能够通过维护一个小型的最近修改的缓存，安全有效地根据旧的字典状态检查陈旧的证明。

贡献。我们提出以下贡献。

1. 一个由矢量承诺支持的认证字典，即使在对抗性的访问模式下，也拥有严格的超额资源使用上限（§5）。
2. 这个字典的版本管理机制，以便它支持并发的事务发布（§6）。
3. 提高词典可用性的技术（§7）。
4. 我们设计的一个实施方案，被整合到一个加密货币的存储后端，并对其进行评估（§8）。
5. 对我们的版本化au-thenticated词典的健全性和完整性属性进行严格的定义和安全分析（§3.2和附录A）。

我们接下来调查了相关的工作（第2节），介绍了系统架构的概况（第3节），回顾了矢量承诺（第4节），然后详细描述了我们的贡献。

2 相关工作

我们的工作重点是短证明，因为带宽往往是加密货币中交易吞吐量的瓶颈[18]。我们专注于效率的具体提高，努力使无状态验证在高吞吐量的加密货币中切实可行：证明规模与交易规模相当（每次操作100-200B），验证器存储成本是
无论对抗行为如何，都会减少一个很大的常数(800×)，而且关键的计算成本也会降低。路径大多是可并行的，允许32核机器每秒处理1000多个操作（§8）。

无状态验证用验证器存储换取带宽（被证明占用）和计算（证明和验证所需）。这些权衡并不能在所有情况下减少系统的总成本。我们现在调查以前的工作，这些工作针对的是权衡曲线上的其他各点。

UTXO模型中的相关工作。在UTXO模型中工作的加密货币围绕交易行为而不是围绕账户组织信息。该系统的状态在

一个特定时间的特征是一个集合：即尚未被用作新交易的输入的交易行动输出的集合。因此，每个交易都需要引用一个先前的交易，并且对于无状态验证来说，提供这个先前交易的输出还没有被使用的证明。在这样的设计中，无状态验证需要动态集数据结构的认证版本（通常称为动态累积器）。已经有许多建议利用Merkle哈希树[32]的某些版本来达到这个目的--例如，见[17, 21, 33, 45, 46]。

基于Merkle树的构造给加密货币增加了大量的网络带宽成本，这可能成为交易吞吐量的瓶颈[18]。例如，128位安全的Merkle证明，对于一个有1亿个条目的动态集，也就是本文写作时比特币UTXO集的近似大小，其大小约为1KB（具体数字取决于使用的底层树结构）。相比之下，一个小交易（例如，在加密货币中转账）可能短至100B，所以使用Merkle树的证明会带来10倍的带宽开销。另一方面，Merkle证明的产生和验证都很迅速。

为了缓解长证明的问题，[21]提出了一些基于证明聚合和树节点的本地缓存的优化方案，将一些状态加回给客户。其他的加密累积器也是合适的；在[20]中可以看到对选项的讨论。特别是，在[12，§6.1]中提出了一个基于未知顺序的组（如RSA和类组）的设计。(这种设计中每个证明都超过1KB，但它允许聚合多个交易的证明)。

基于账户的模型中的相关工作。由于基于账户的加密货币维护着账户（通常由公钥识别）到账户数据（如余额）的映射，因此使用认证字典来减少客户端状态是很自然的。大多数认证字典的构造[15, 24, 34, 37, 40, 50]是基于一些
Merkle树的变种[32]，也存在证明时间长的问题，对于一个简单的交易，涉及到访问两个钥匙的交易，会有10-20倍的带宽开销。
词典（例如，来源和目的地账户）。

一些不基于Merkle树的认证字典的构造是静态的，没有对数据进行有效更新的规定（例如，[27]）。一些早期的不基于Merkle树的动态构造（例如，[39]和[19]、§4]）要求验证者对证明者保守秘密，以防止证明者作弊。[1]因此，它们不适合加密货币应用，因为加密货币验证者是公开的，因此不能信任秘密。

Concerto[4]将验证推迟到最后，产生一次线性成本（在字典大小中），而不是一直有小成本。这种方法不适合加密货币，因为交易需要实时处理。

[1] 确切地说，[39]可以在没有验证人秘密的情况下工作，但如果证明的大小保持在较小的范围内，字典的更新会变得很昂贵（根据[39，表2和表3]，证明大小在1KB以下时，每次更新大约需要几秒钟）。

Boneh, Bünz, and Fisch [12, §5.4, §6.1]提出了第一个不需要验证者秘密和不使用Merkle树的验证字典；相反，他们依赖于基于因子（或基于类组）的累积器和向量承诺。他们还建议（但没有调查）使用这种字典来实现无状态验证的成本。他们的构造中的证明长度超过1KB，因此与Merkle树（在128位安全时）相当。然而，有可能将多个操作的证明批量化为一个恒定大小的证明。后续工作[1，49]以重要的方式改进了[12]的各个维度，增加了功能和效率；但是，证明的长度仍然超过500字节。

Tomescu等人[48]解决了一个有点不同的问题，即使用双耳配对的仅附加认证的集合和字典；他们的证明（对于不同于简单读写的操作）在合理的安全参数和账户数量下有几千字节的长度。

Chepurnoy等人[17]提出了一个被称为EDRAX的优雅解决方案，避免了认证字典。EDRAX将连续的整数（数组中的索引）作为账户标识符，从而将必要的数据结构从一个认证的字典简化为一个矢量承诺。这种方法增加了新账户注册的复杂性：EDRAX要求每个新的公钥通过初始化交易注册，而不是简单地发布一个交易，将钱发送到一个新的公钥（如Aardvark）（防止使用该交易作为拒绝服务的载体在[47，§4.2.4]中讨论）。这个公钥的其他交易必须知道索引，并且在注册发生之前不能被处理；同时，注册在计算上是很昂贵的，我们在本节后面讨论。

公共参数和承诺大小。一些基于配对的矢量承诺方案需要一组可信的公共参数。在已知的基于配对的矢量承诺中，这些参数至少与矢量长度成正比。EDRAX（及其改进版[47]）要求公共参数与系统中的账户数量成正比，因为所有的数据都存储为一个单一的向量。相反，Aardvark使用多个向量（共享相同的参数）。这种方法避免了对系统中账户数量的*先验*约束，不需要冗长的参数生成。然而，Aardvark的验证器需要存储多个向量承诺，而不是只有一个。对于合理的矢量长度，这允许验证器存储总状态的一小部分。

用SNARKs缩减长证明的通用性。原则上，任何认证数据结构的长证明都可以通过应用简洁的非交互式知识论据（SNARKs）缩减到恒定大小，SNARKs是通用工具，可以适用于任何计算。

目前最先进的SNARKs[26]的证明相当短--只有配对友好椭圆曲线上的三个点，或者128位安全的大约144字节。这种方法

是在Zcash[8]中对基于UTXO的加密货币采取的（注意Zcash将隐私作为其主要目标，这给设计引入了额外的复杂性）。EDRAX[17]建议它用于基于账户的加密货币。SNARK验证是相当高效的，需要的椭圆曲线点乘法的数量与被证明的声明的大小成正比，并且只需要三个椭圆曲线对的一个乘积[26，表1和表2]。基于SNARK的方法的缺点在于证明的成本。计算一个SNARK是很耗时的。例如，据报道，证明Zcash交易的声明（公认的复杂）的SNARK需要3秒[9，§1]，而EDRAX的矢量承诺证明的SNARK需要77秒[36]。(关于Aardvark和EDRAX的更详细的比较见第8.1节）。

有了近乎恒定的验证时间的优势，一些建议（通常被称为 "Rollups"）将多个交易批在一个证明中（例如，见[16，23，31]）。虽然大大节省了验证者的时间，但这种建议会产生很高的验证者延迟，因为验证者必须在产生证明之前收集许多交易，而且每个交易的证明成本是一秒钟的数量级[28，图7]，很难并行化。

SNARKs是一个活跃的研究领域，基于SNARK的无状态验证方法可能会有所改进，在某些情况下可能会超过Aardvark。特别是，Ozdemir等人报告说[35，图6]，一个单一的Merkle证明（当哈希函数被选择为对SNARK特别友好时）可以在大约100ms内转换成SNARK。

谁提供证明？任何无状态估值的方法都必须解决谁负责维护外包数据并为每笔交易提供必要的证明的问题。我们重申，证明不是静态的：即使一个用户的账户数据没有变化，该数据的正确性证明也会随着其他账户的变化（并导致承诺的变化）而变化。

Zcash和EDRAX要求用户存储证明，并通过与加密货币的交易同步来保持它们的新鲜。隐含的是，客户依赖于不受信任的存储，因为如果他们在上线时错过了中间的交易，他们必须从存储中获得它来赶上。

这种设计给EDRAX的新账户注册带来了前面描述的问题。没有人可以为一个还没有附加到账户的指数保持最新的证明。一个新的用户必须通过从头开始阅读每一笔交易来计算这个证明，或者依靠一个知道整个当前矢量的服务器来提供证明[36]。无论哪种方式，其成本至少与账户数量成线性关系。Aardvark使用多个短矢量承诺的方法也可以应用于EDRAX，以减少这种成本。

由于EDRAX是基于具有长公共参数（与系统中的账户数量成正比）的矢量承诺，EDRAX设计的一个重要特点是确保每个用户只需要公共参数的一个小子集来同步证明。Tomescu等人[47]im-

证明这个矢量承诺，实现恒定大小的证明，并且只需要相关操作的公共参数的恒定大小部分。

在基于账户的模型中，这种方法的一个缺点是，它限制了交易可以对接收方账户造成的变化种类。交易的发送者不能提供接收者账户状态的证明（无论是旧的还是更新的），这意味着交易可以对接收者账户进行的唯一修改是那些不需要知道账户状态的修改。在不知道承诺状态的情况下修改它需要某种承诺的同构性。EDRAX依靠其矢量承诺的加法同构性来实现向未知状态的账户添加资金，所以交易除了向收款人添加资金外，什么也做不了。因此，不支持更复杂的操作（例如，智能合约）。同样，基于UTXO的Zcash只能给收款人一个交易输出，而不是以更普遍的方式修改收款人的状态。相比之下，Aardvark将计算和提供证明的问题外包给不受信任的档案节点。这种设计使我们能够对账户信息进行任意的更新。

发件人和收件人的信息。

失序的交易处理和陈旧的证明。重新呼吁，交易是异步发布的，然后由加密货币共识机制进行or- dered。由于账户数据的正确性会随着其他账户的变化而变化，所以当带有证明的交易到达验证者时，证明可能会变得陈旧。处理陈旧证明的机制是必要的，以确保交易不会被不必要地拒绝，这可能会严重限制加密货币的吞吐量--特别是如果计算和传输证明的时间很长。同时，这些机制不能过于放任，因为它们必须防止重复消费。因此，验证者不能简单地接受由陈旧的证明所验证的账户状态，即使它对最近的承诺值是有效的，因为自证明发布以来，该状态可能已经改变。

大多数相关工作都没有解决这个问题，而是假设证明在过时之前就已经到达验证者手中（例如，[17]和[8，§8.3.2]）。Aardvark的缓存机制解决了这个问题（§6）。

3 概述

在一个加密货币中，一组*验证者*共同维持着状态。当前的状态是由一个众所周知的（"创世"）初始状态定义的，并由一连串的原子*交易*修改。客户端提交交易，验证者执行一个协议来验证这些交易，如果它们是有效的，就把它们追加到序列中。这个协议被去掉签名以确保共识，这样所有验证者都同意相同的公共交易序列。交易被称为

当它们被追加到这个序列中时，就会被*确认*。

为了决定是否接受一项交易，验证者需要知道系统的当前状态。例如，如果Alice发出的交易花的钱比她的多，验证人必须拒绝这个交易；为了做到这一点，他们需要知道Alice的当前余额。

Aardvark使验证者能够使用系统状态，而不要求他们存储它。相反，不受信任的*档案馆*维护这个状态，而验证者只维护一个小的承诺，这个承诺随着状态的变化而变化。为了发布一个交易，客户查询一个档案馆，以获取与交易有关的数据。档案馆返回数据和被称为*上下文的*验证信息[2].客户端将交易连同数据和上下文一起提交给验证者，验证者根据承诺检查数据和上下文，并验证交易。一旦该交易被确认，验证者使用上下文来更新他们的承诺，而档案馆则更新他们的系统状态副本。

请注意，如果一个交易T是在状态s_1下发出的，并且带有对应于s_1的上下文，那么其他交易可能会在s_1之后，但在验证者进入处理之前被确认。T.在验证者处理T时，他们的承诺可能对应于后来的某个状态s_2。为了处理T，验证者必须能够使用来自s_1的上下文来验证T，并在T被确认后根据T的结果更新他们的承诺（这些承诺对应于s_2）。Aardvark解决了这个失序交易的挑战处理，只要在s_1和s_2之间确认的交易不超过τ（其中τ是一个系统参数）。

系统组件。我们的系统是建立在向量承诺上的，在第4节中有所描述。我们首先使用向量来实现基本的二进制操作（下面的Get、Put和Delete）（§5）。接下来，我们通过为二维码开发一个版本计划，使Aardvark能够不按顺序执行事务（第6节）。然后，我们为Aardvark引入一种机制，即使存档失败也能继续提供服务（第7节）。

3.1 交易接口

Aardvark提供了一个*字典*接口，*将键*（账户标识符）*与值*（账户余额和其他信息）联系起来。在一个事务中，客户通过对字典发出一连串的操作来读取和修改键值映射，从而与状态进行互动。

具体来说，Aardvark支持以下操作：Get, Put, and Delete。Get(k)返回与键k相关的值v或一个特殊的值，表示k不存在。Put(k, v)将一个新的值v与k关联，覆盖任何旧值。Delete(k)使k与任何值脱离关系。

在Aardvark中，要么所有交易的操作都执行、或不做。一个事务指定一组键（一个位串

[2] 我们用 "contexts "而不是 "proofs "来表示认证的字典，以避免与底层矢量承诺中的证明相混淆。矢量承诺证明是语境的元素。

识别一个单一的值）和涉及这些键的操作列表。一个事务所引用的所有键必须被预先指定。特别是，诸如动态键读取的操作是不被支持的。(更多细节见第6.3节）。

让alice和bob表示两个余额的钥匙。(为简单起见，假设Get对缺失的钥匙返回0。)那么，从alice到bob的p个单位的钱的转移可以通过交易来实现，用伪码表示。

```
with Transaction(alice, bob) as txn:
    a = txn.Get(alice)
    b = txn.Get(bob)
    assert a >= p
    if a-p == 0:
        txn.Delete(alice)
    else:
        txn.Put(alice, a-
    p) txn.Put(bob, b+p)
```

如果验证器存储系统状态，他们将有足够的信息来处理交易。因为在Aardvark中它们没有，所以客户要求档案馆提供Con-Based。
交易中每个字典操作的文本。给定一个键和一个操作，档案馆计算出上下文σ。对于交易中的每个字典操作，客户
提供从档案馆获得的上下文。

3.2 安全问题

加密货币依赖于一定比例（例如大多数）的验证者忠实地遵守其协议的规则。Aardvark假设这些相同的验证者也遵循其协议。相比之下，档案馆和客户被允许任意偏离正确的行为。

Aardvark的目标是提供与理想字典相同的功能，以应对计算上受限的对手。这种功能在Aardvark中表现为两种方式：健全性和完整性。在这里，我们非正式地陈述这些属性，并说明其正确性。附录A对它们进行了形式化的证明。

健全性。对验证器来说，每一个事务的效果与验证器拥有大的持久性存储并自行存储整个状态，没有任何存档的情况下是一样的。这一属性在§A.1.2中得到了精确的规定。在高层次上，这个属性是通过归纳法保证的，如下所示。承诺最初通过设计反映了创世状态。如果承诺反映了当前的状态，那么验证者将只接受具有正确数据的交易，并将更新其承诺以正确反映新的状态。我们在§A.3中正式说明了这个证明，它依赖于底层向量承诺的安全性。

完备性。验证者将接受一个交易，只要它包括来自档案馆的正确上下文，并且在这些上下文之后没有太多的其他交易被确认。

文本被创建。这个属性在
§A.1.1，通过检查我们的设计可以看出。

3.3 可利用性

Aardvark在一个客户和档案可能是恶意的环境中运作。Aardvark的设计有两个方面使它能够在这种情况下提供可用性。

对抗性交易下的效率。在对抗性修改模式下，验证者的存储成本被状态大小的一小部分所约束，这使得系统可以收取存储费用以防止拒绝服务攻击。具体来说，加密货币可能要求每个账户持有固定的最低余额，因此这种攻击需要大量的持续投资。§第8节以分析的方式评估了这个上界。

无存档操作。即使没有存档功能是正确的，客户也可以监控公共交易序列以保持任何给定密钥的当前上下文。§第7节描述了客户机如何在不与档案库交互的情况下同步其上下文。

4 背景：向量承诺

为了解释向量承诺，我们回顾一下哈希函数。一个加密散列函数H，给定一个输入X，产生$h=H(X)$。因为找到另一个$X' \neq X$是不可行的。
这样，$h = H(X')$，我们可以说，h是对"承诺"的X：X的存储可以外包给一个不受信任的服务器，因为我们可以通过比较哈希值和h来验证不受信任的服务器向我们发送的数据。

矢量承诺可以被看作是加密哈希函数的一般化。一个矢量承诺方案不仅可以承诺一个输入X，而且可以承诺整个数据阵列V：承诺程序产生一个对矢量V的简短承诺c。与单纯的哈希函数不同，一个矢量承诺方案
如果给定一个证明π_i，其中π_i是由知道V的人提出的，那么知道c的人不仅可以验证整个V，还可以验证单个元素$V[i]$，而不用看到其他元素。承诺和证明
当一个值发生变化时，是可以有效地更新的。

我们首先描述了矢量承诺方案所需的接口和安全属性，然后讨论我们使用的具体矢量承诺方案。

4.1 界面和安全属性

我们考虑在参数生成时固定阵列大小的方案、[3]的方案，其黑盒接口如下。

[3] 有一些矢量承诺方案不需要这样做，但它们对我们的目的来说不够有效。

$\text{ParamGen}(B) \rightarrow \text{pp}$

$\text{Commit}_{\text{pp}}(V) \rightarrow c$

$\text{Open}_{\text{pp}}(V, i) \rightarrow v_i, \pi$

$\text{Verify}_{\text{pp}}(c, i, v_i, \pi) \rightarrow \text{ok}/\perp$

$\text{CommitUpdate}_{\text{pp}}(c, (i, v_i, v'_i)) \rightarrow c'$

$\text{ProofUpdate}_{\text{pp}}(\pi, j, (i, v_i, v'_i)) \rightarrow \pi'$

ParamGen生成的参数pp可以用来承诺一个大小为B的数组。这些参数必须由一个可信的程序生成（例如，通过多方计算），不泄露所使用的秘密。一组参数可以用于多个数组，每个数组的大小为B。这些参数pp被所有其他算法使用；当下标pp不会引起歧义时，我们会省略它。

Commit为一个B级长度的可变数据向量创建一个加密承诺。[4]Commit是确定的：如果v_1和v_2是两个B大小的向量，并且$v_1 = v_2$，则$\text{Commit}(v_1) = \text{Commit}(v_2)$。我们依靠的是以下的确定性致力于确保验证器的状态是一致的。

Open创建一个加密证明，证明一个特定的向量元素v_i（存在于向量v的索引$i < B$）在$\text{Commit}(V)$中被承诺。Verify检查一个证明是否有效。

CommitUpdate返回$\text{Commit}(V')$，其中V'得到的是通过将第i个条目从v_i改为v'。同样，ProofUpdate为这个V中的元素j返回证明$\text{Open}(V', j)$。请注意，CommitUpdate和ProofUpdate只需要V的一个元素，与Commit和Open不同。

承诺的*约束*属性[5]确保证明除v_i以外的任何值都在索引i处，在计算上是不可行的。[6]

4.2 具体选择：利伯特和容的矢量承诺

对Aardvark来说，底层矢量承诺的具体选择并不重要，只要当矢量中的一个元素发生变化时，承诺可以被有效地更新。然而，如前所述，我们对承诺的特殊选择是出于对效率的要求。特别是，带宽成本是限制系统吞吐量的一个因素。我们使用Libert和Yung[30]的向量承诺。他们有最小的证明：对于128位的安全，只有48个字节，或者对于合理的数据大小和安全参数，比Merkle证明短一个数量级。此外，正如Gorbunov等人[25]所示、

[4] 可变长度的数据总是可以在应用Commit之前通过 散列转换为固定长度的数据。

5Aardvark不需要承诺 方案的隐藏属性，因为档案馆存储的备份数据是公开的。

6从技术上讲，我们只需要在c是诚实产生的情况下保持约束力；然而，许多矢量承诺方案确保更强的约束力，即即使c是由对手产生的。

多个证明（即使是不同的承诺）可以被归纳为一个48字节的值，并可以被一次性验证。它们依赖于基于配对的密码学，这在过去十年中得到了采用和部署[41]。

这些矢量承诺需要公共参数pp在B中呈线性关系（参见[25，§4.1]关于如何生成这些参数的讨论）。承诺和证明都需要在B中呈线性关系的时间来创建，并且需要恒定的时间来更新（参见表3），而验证在恒定时间内运行，但需要对相对昂贵的配对进行评估。§8.1将我们选择的矢量承诺与EDRAX使用的承诺进行了评估。

这些考虑影响了我们的设计。首先，B的选择允许Aardvark在节省存储和计算开销之间进行权衡。增加B*可以*减少存储成本的一个恒定因素，并增加证明生成成本的一个恒定因素。

第二，验证成本仍然相对较高，并不能通过缩减B*来*缓解。虽然证明生成工作是由不受信任的档案库完成的，并且可以在许多不受信任的机器之间分配（例如，在一些具有廉价存储的云提供商），但证明验证必须由估值者完成，所以它位于关键路径上。因此，为了管理商业成本，Aardvark必须允许证明验证在独立线程上并行执行，并尽量减少所需的验证操作数量。

5 认证的字典设计

Aardvark维护了一个从键（任意的比特串）到其值的认证映射。由于向量承诺提供了一个数组接口，我们必须找到一个有效的数组编码，来实现字典的映射。我们维护这种映射的方法是将所有的键-值对以任意的顺序存储在大小为B*的*数组中（记得从§4.1，B是矢量大小）。为了证明一个密钥是与在给定的值上，验证器在包含键值对的任何索引上运行 Open，如果该键在字典中正好出现一次，则产生一个正确的结果。修改一个键的值需要运行CommitUpdate。

不太明显的是，验证者如何证明一个密钥*不*存在。如果验证者在地图平移中谎称某个键不存在怎么办？如果我们解决了这个问题，我们也可以确保每个键最多出现一次，因为作为插入证明的一部分，验证者将证明在插入之前该键不在字典中。

Aardvark通过使用两种不同的键值对来实现这一不变性。首先，Aardvark在长度为B的向量*槽*中以任意的*顺序*排序来提交这些配对。因为这个排序是任意的，所以我们可以一直追加到最后一个向量的末尾。通过在删除时急切地压缩向量，我们将加密操作的数量和空空间的数量都降到最低。第二，Aardvark承诺对其密钥进行独立的词典*式*排序，允许证明一个密钥的ab-b。

词典中的逻辑。我们通过将每个键与它的词法继任者一起存储来做到这一点。这样一来，Aardvark就会以递增

的词法顺序提交一个键的链接列表。

具体来说，每个槽包含一个三联体（k, v, succ(k)）。鉴于所有钥匙的集合，我们对任何钥匙k的继任函数succ(k)定义如下：

1. 如果k是字典中最大的键， succ(k) 就是字典中最小的键。
2. 否则， succ(k)是字典中大于k的最小的键。

我们可以用类似的方式来定义一个键的前身。

证明k映射到v需要在包含槽（k, v, succ(k)）的索引处打开向量承诺。证明k不在字典中涉及打开
在包含槽位的索引处摄取向量承诺
(k_0 , v_0 , succ(k_0))，其中$k_0 \neq k$， succ(k) = succ(k_0)。

初始化。我们的字典是用向量组合参数初始化的，参数化后的桶大小为B，如上节所述。此外，它还用一个单一的键和值进行初始化，因为它必须保持内部的不变性，即在任何时候至少有一个键存在。[7]验证器必须计算初始承诺值，因为它不能相信档案馆会正确计算它。

5.1 毗连的槽位包装

实现我们的字典的一个挑战是，用户可能希望插入许多任意的键，然后随后重新移动这些键。Aardvark必须有效地扩大和缩小这些修改，而这些修改在本质上可能是敌对的。我们对Aardvark的分配方案保证了一个有界限的最坏情况下的存储成本，而不管访问模式如何。Aardvark的插槽被连续分组为一个个的*桶*

大小为B：第ℓ个桶存放编号为[$B\ell$, $B\ell$ +]的槽的散列值。B).将*第ℓ个*桶中的数据称为D[ℓ]。对于每个ℓ，档案馆存储D[ℓ]，而验证器存储Commit(D[ℓ])。验证器也存储字典的总大小，s。让D表示所有D[ℓ]的序列，并定义*字典*的*承诺*是所有向量承诺的序列，以
所有的桶和值s。

要修改一个槽，存档者要计算该槽在槽排序中的绝对索引，以及该槽在该索引的当前值的证明。这些数据允许验证者在原地修改槽。

```
def slot_write(db: handle, i: index、
               old: slot, new:
slot): bucket, offset = i/B, i%B
c0 = db.get_commit(bucket)
c1 = CommitUpdate(c0, (offset, old, new)
db.set_commit(bucket, c1)
```

[7] 这可以通过一个永不删除的哨兵钥匙来实现。

```
A = Slot(key="aardvark", val=2, next="cat")
C = Slot(key="cat", val=3, next="dog")
{A, C}                                      5
```

```
A = Slot(key="ardvark", val=2, next="bear")
B = Slot(key="熊", val=5, next="猫")
C = Slot(key="cat", val=3, next="dog")
{A, B, C}
```

图1：Aardvark的密钥插入包括以下步骤。(1) 钥匙被插入到字典的最后一个槽中。(2) 其前任被验证，新槽的下一个指针被附加到前任的下一个指针上。(3) 其前任的下一个指针被更新为插入的键。请注意，这里显示的操作只需要对矢量承诺进行一次更新操作（更新前任的下一个指针），因为尾巴是由验证器存储的，并没有承诺。

为了严格控制字典的空间开销，Aardvark的打包方案保持了一个不变的原则：估值器最多存储⌈s/B⌉矢量承诺，再加上一个小的、恒定数量的元数据。当一个新元素被插入时、它被添加到字典中的最后一个槽（在s）。

```
def slot_append(db: handle, new:
    slot): s = db.size() + 1
    db.set_size(
    s) if s%B
    == 1:
        db.push_commit(Commit([new]))
        return
    bucket, offset = s/B,
    s%B c0 =
    db.get_commit(bucket)
    # zeros是重复256次的 "0 "位 c1 =
    CommitUpdate(c0, (offset, zeros, new))
    db.set_commit(bucket, c1)
```

同样，当一个元素被删除时，它被最后一个槽（在s）的元素所覆盖，并且最后一个槽被清除。

5.2 验证读和写

为了证明一个密钥k与一个特定的值相关联，一个档案馆找到了该密钥的槽，并计算出它与Open的成员关系证明。然后，档案馆将槽、它的索引和它的证明作为一个*上下文*目标传送给验证者。对于验证器来说，鉴于其上下文，将密钥重新分配到一个新的值是很直接的


```
Put("熊", 5)
```

：验证上下文，使用索引来定位槽，并用其新值覆盖槽。更新槽是由slot_write实现的，而上下文验证则需要执行Verify。

```
def verify_ctx(db: handle, ctx: context):
    i, val, pf = ctx.index, ctx.slot, ctx.pf
    bucket, off = i/B, i%B
    c0 = db.get_commit(bucket)
    return Verify(c0, off, val,
    pf)
```

证明一个键在字典中不存在也是类似的，只是档案馆要证明前任的槽的存在。因此，Get只是通过检查我们处于这些情况中的哪一种来实现。

插入一个新键k的映射涉及到向最后一个桶追加一个持有k及其值v的槽。此外，我们保留键的词法排序，类似于链接列表的插入，这需要k的前身的背景。因此，Put可以被实现为：。

```
def Put(db: handle, key: key, val: value,
        ctx: context):
    assert verify_ctx(db, ctx)。
    assert ctx.slot.key <= key < ctx.slot.next
    如果ctx.slot.key == key:
        old, new = ctx.slot, ctx.slot
        new.val = val
        slot_write(db, ctx.index, old, new)
        return
    new_last = Slot(key=key, val=val、
                    next=ctx.slot.next)
    slot_append(db, new_last)
    old_pred, new_pred = ctx.slot, ctx.slot
    new_pred.slot.next = key
    slot_write(db, ctx.index、
               old_pred, new_pred)
```

最后，Delete是Put的逆运算，需要三个文本：k本身的文本，其前身的文本（用于链接列表的删除），以及字典中相对于*顺序*的最后一个槽的文本（因为删除一个键需要用最后一个槽的内容来覆盖其槽的内容）。

5.3 尾部操作

插入和删除操作都会影响到*尾部*的*桶*，即字典中的最后一个桶。通过对这个桶进行批量加密操作，Aardvark降低了维持其承诺的成本，并避免了插入和删除槽的需要，为最后一个槽提供了一个背景。

具体来说，不保持对尾部桶的承诺。添加槽位的投放操作不会导致立即创建一个新的向量承诺。相反，新的槽直接存储在验证器上，直到积累的槽超过B，这时验证器会执行一次承诺操作来生成新的承诺。

同样的，对于删除操作，尾部桶中没有承诺的槽会在没有CommitUpdate操作的情况下被移除。一旦没有这样的槽，验证器就会解约到

通过获取最后一次承诺的预像，并执行一次承诺操作，就可以一次完成整个桶的验证。从这里开始，验证器再次将桶的槽位存储在预像中，而不进行承诺。

由于单次解约的带宽成本相对较高，但随着时间的推移会摊薄，验证器将桶的一些后缀的预像缓存在一个*尾部缓存*中，并在后台同步这个缓存。随着插入操作的积累，验证器忘记了旧的桶，随着删除操作的积累，它从档案馆请求桶。如果验证器缺乏对尾部水桶的预像，它将拒绝为后续的删除请求提供服务，直到它能够重新同步。为了减少插入和删除交替进行时的激动，验证器保持最小的尾部缓存大小为L，最大的尾部缓存大小为$2L$。

6 订单外的交易处理

第5节中的字典设计在包含写的事务的每次提交时都会改变状态，使针对旧状态的证明无效。具体来说，执行一个写到某个桶的事务将使任何同时发出的使用相同桶的事务的证明失效。在一个高吞吐量的系统中，证明可能会经常失效，这就要求客户或档案馆不断地重新计算并重新提交待处理事务的证明。这个问题在大负荷下会恶化，因为积压的交易会进一步增加重新计算证明的成本。此外，攻击者可能会通过不断修改用户的密钥桶邻居来降低对用户的服务，使其旧的证明失效。不幸的是，我们的评估表明，ProofUpdate对于我们的矢量承诺来说是昂贵的（§8.1）。Aardvark通过完全消除更新陈旧证明的需要来解决这个问题。相反，Aardvark采用了一个*版本*系统，允许它接受一个带有陈旧证明的交易，即使是在并发操作修改状态的情况下。该交易所参考的。

首先，验证器给它执行的每个事务分配一个序列号，把它附加到已完成的事务序列中。依次应用每个事务产生当前的字典状态。因此，这个序列的每个前缀决定了一个特定序列号的字典状态的单一*快照*，而序列号列举了字典的版本。

当发布一个事务时，客户声明它可以执行的最小版本t_0。附在交易上的上下文是指截至t_0的字典快照。交易的执行时间不得超过t_0 +τ版本，其中τ是系统的*最大交易寿命*，是初始化时设置的全局常数。较大的τ值允许旧的交易证明被系统接受更长的时间。当词典处于高负荷状态，客户的交易可能在执行前就过期了，迫使其上下文被更新。

在交易被重新提交之前，较大的τ可以减少系统拥堵时的额外工作。

其次，验证器持有的字典承诺不是最新的版本，而是τ年前的*基本快照*。对于最古老的状态和最新的状态之间的每一次交易，验证器都会对交易引起的变化进行缓存，所以较小的τ可以减少缓存过载。这些变化会修改向量的承诺。

桶，所以验证器也会缓存修改后的向量组合。因此，验证器可以通过检查任何不超过τ版本的快照来验证证明。

对照缓存的修改过的向量承诺。

6.1 完整性和上下文版本控制

客户端开始查询存档，以获得执行事务所需的一组上下文，其中包括获取、投放或删除操作。对于每个操作，档案馆会返回验证器所要求的槽，它们在当前版本的字典中的索引，以及它们相对于其包含的桶的开口，还有当前的

$_0$当验证器收到一个交易以及后来的版本$t \geq t_0$的上下文时，验证器必须同时确定有关的槽的值。

交易的运作，同时确定更新其字典承诺所需的变化。

首先，验证器对上下文进行验证，检查它们是否确实对应于在版本t_0时有效的槽。这要求验证器知道截至t_0的矢量承诺，因为档案馆对该版本开放承诺。其次，一旦验证器有了正确的t_0的槽，它就会更新这些槽，使它们对t来说是正确的。这需要验证器知道在t_0和t版本之间发生的更新。一旦验证器有了t版本的所有槽，它就可以验证和执行交易，这将产生对当前快照的新更新。

为了实现完整性，验证器必须处理所有版本t_0的证明，其中$t - \tau \leq t_0 \leq t$。因此，验证器的承诺对应于版本$t - \tau$时的基础快照，并且验证器必须存储最后的τ个事务。每当验证器对状态应用一个新的事务时，它就会保存新的事务，并将最旧的事务标记为可删除的事务；旧的事务在被收集垃圾之前被异步地应用于承诺。

6.2 缓存

如前所述，验证器必须验证为版本$t - \tau$和版本t之间的任何快照产生的证明，这意味着它必须计算向量commit-对应于该版本的承诺。因为验证器最终必须计算所有的向量承诺，以便更新对基础快照的承诺，Aard-vark维护了一个*向量承诺deltas*的缓存。这

对于t -τ和t之间的每个版本t'，新旧向量承诺对应于被序列号为t' 的交易所修改的每个桶。该缓存使验证者能够在不需要验证的情况下对证明进行验证。为每个事务运行CommitUpdate操作。

就像向量承诺一样，验证器也可以类似地缓存*槽位deltas*和*键位deltas*，以加快槽位的查找（通过索引）和键值对的查找（通过键），分别保存这些槽位和键的新旧值。图2说明了这些缓存如何使验证器在事务中处理操作。

确定性。Aardvark要求更新是确定的，并且上下文验证在验证者之间是一致的，以便他们计算出匹配的承诺，这在加密货币中特别重要。更新是通过强制执行规范的顺序来实现一致性的，例如在应用deltas之前对键进行排序，并以明确的顺序处理键的插入和删除。

区块批处理。为了减少维护deltas的内存开销和搜索deltas的键的成本，Aard-vark支持将交易批量化为*块*。通过区块提交的优化，版本（和τ）可能指的是区块序列号，而不是事务序列号。一旦一个交易块被处理，Aardvark
合并该区块中的所有删除操作。通过匹配区块的密钥插入操作和删除操作，Aardvark重新减少了所需的加密操作总数，并加快了对基础快照的查询和更新。

6.3 交易表达能力

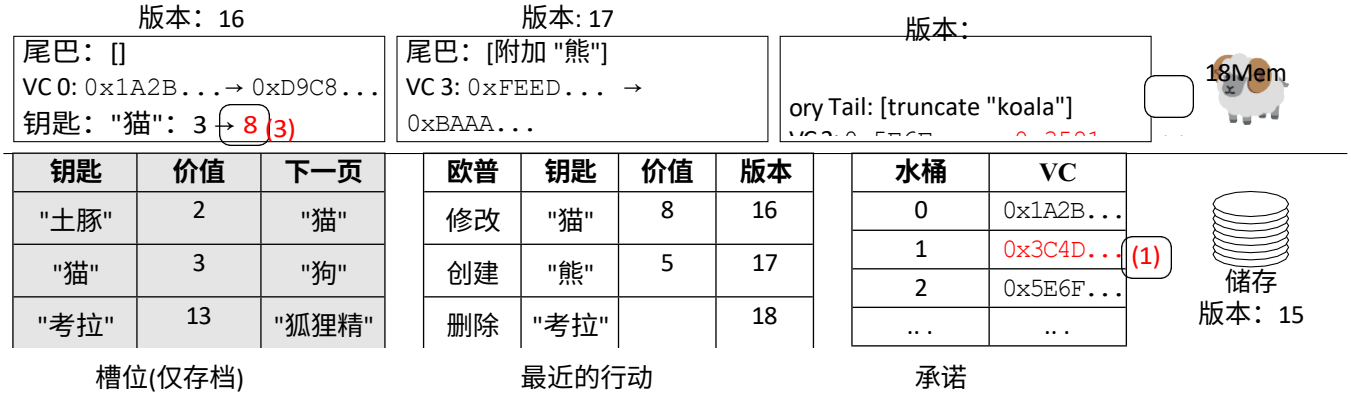
我们选择的版本管理方案允许Aardvark的交易是相当有表现力的。例如，对数值进行换算的事务，如加法或最大函数，在Aardvark中很好地交错进行。同时，认证的字典拥有一个固有的限制。一旦创建了一个上下文，所有的键都必须是固定的。

为了说明问题，假设一个密钥k_1 的值是另一个密钥k_2 ，而某个交易T执行了Get(Get(k_1))。T没有办法计算属于密钥k_2 = Get(k_1)的证明，因为其他交易可能在T发出后但在T被确认前执行Put(k)。[1]

7 无档案操作

上面介绍的字典设计依赖于档案的可用性：如果档案失效或拒绝为客户提供服务，客户就不能执行所有操作。然而，只要稍加修改，大多数操作都可以做到*无档案*：它们可以与档案解耦，即使所有档案都不可用，也可以执行。[8]

[8] 无档案操作的另一个好处是，它们减少了档案的稳态计算负荷。这一点，在扩展的时候是很重要的



钥匙	价值	下一页
"土豚"	2	"猫"
"猫"	3	"狗"
"考拉"	13	"狐狸精"

欧普	钥匙	价值	版本
修改	"猫"	8	16
创建	"熊"	5	17
删除	"考拉"		18

水桶	VC
0	0x1A2B...
1	0x3C4D... (1)
2	0x5E6F...
.. .	.. .

图2： 不同的Aardvark交易中的三个Get（τ = 3）：(1) Get("ardvark", σ= {val: 2, index: 1, version: 17}) → 2, (2) Get("walrus", σ= {val: 5, index: 3, version: 18}) → 34, and (3), Get("cat", σ= {val: 3, index: 0, version: 12}) → 8.对于(1)，钥匙本身和它的桶最近都没有被修改过，所以档案馆的证明保证对向量是有效的
承诺在验证器的存储中。对于（2），数值本身最近没有被修改；但是，一个删除操作重新定位了它。因为它的证明是在重新定位后创建的，所以该承诺是在一个矢量承诺delta中。对于(3)，该值最近被修改过，所以证明是不必要的；结果，字典可以简单地查找存在于键delta中的值。

具体来说，考虑一个客户端在版本t时拥有某个键k的正确上下文，然后观察字典的更新到版本t+1。客户端上下文的更新必须覆盖（或删除）k，修改k的一个桶邻，或将k移到另一个槽（由于Delete）。在第一种情况下，只有当k被删除时才需要证明更新，其中
就足以让客户端存储k的前任的上下文。这
第二种情况对应的是一个简单的ProofUpdate操作。最后，在第三种情况下，删除操作包含新槽的上下文，客户端可以存储。

因此，一个保持钥匙和/或其前身的新鲜证明的客户端可以发出任意有效的获取、投放或删除操作，而无需查阅档案。此外，证明
更新只需要每τ个序列号发布一次。有一个例外：在某一点上，一个验证器可能会运行
由于发生了太多的删除，它的尾部缓存被删除。在这种情况下，不能执行后续的删除，因为尾巴不能被移动以填补新的槽位。

8 评价

我们的评估回答了以下问题：

- *矢量承诺*：我们选择的矢量承诺的性能特点是什么，它们与其他替代方案相比如何？
- *存储的节省*：与存储所有数据的存档相比，验证器需要多少存储？
- *带宽成本*：Aardvark的语境会带来哪些额外的网络开销？
- *处理开销*：验证人和档案馆所需的处理费用是多少？

在用户群中，由于开放涉及到一个非微不足道的计算（§8.1）。

表1： VC操作延时（平均值±SD μs）

运作	土豚	Merkle Tree	艾德雷克斯 不含 SNARK
承诺	40262 ± 129	1317 ± 4	-
开放式	40277 ± 444	< 1	-
核实	3707 ± 10	9 ± 0	3131 ± 9

表2： VC对象大小（字节数）

宗旨	土豚和 [47]	Merkle Tree	爱德瑞克斯 不含 SNARK	爱德瑞克斯 与 SNARK 合作
淘宝网。	48	32	64	64
证明	48	320	640	192

我们给出了存储和带宽成本的闭合式表达。对于计算成本，我们提出了对一个原型实现的经验评估，该原型可在 https://github.com/derbear/aardvark-prototype。

8.1 向量承诺

我们在我们的向量承诺实现中，在基本的Merkle树中，以及在EDRAX中使用的实现中评估了这些操作，包括有无SNARK [17]、§4.1].我们与[47]的向量承诺相比较，分析了

表3： VC曲线操作（标量乘法，配对）。

运作	土豚和[47]	爱德瑞克斯不含SNARK	爱德瑞克斯与SNARK合作
承诺	$(O(B), 0)$	$(O(B), 0)$	$(O(B), 0)$
开放式	$(O(B), 0)$	$(O(B), 0)$	SNARK
核实	$(O(1), 2)$	$(O(1), \log B)$	$(O(1), 3)$
承诺更新	$(O(1), 0)$	$(O(1), 0)$	$(O(1), 0)$
证明Update	$(O(1), 0)$	$(O(\log B), 0)$	SNARK

从理论上讲，因为没有实现。

我们在Rust[3]中实现了我们的向量承诺方案，使用的是代数运算的配对库[2]（这是Bowe的库[14]的一个分叉）。[9]我们将我们的实现与EDRAX[52]的实现进行比较。尽管EDRAX提出了一个带有SNARKs的扩展（第2节），但其实现却不可用。此外，该实现既不支持Commit，也不支持Open，而EDRAX并不要求这一点，因为客户端会递增地更新证明和承诺。

我们在一个拥有英特尔至强白金2.5GHz CPU的`m5.metal`亚马逊EC2实例上进行单核微基准测试，L1、L2和L3缓存的大小分别为32KiB、32KiB和1MiB。为了计算操作吞吐量，我们执行100次迭代来预热机器状态，然后进行1000次测量。我们预先计算我们的矢量承诺中基点的功率，内部节点
的梅克尔树，以及EDRAX中的更新公钥。我们设定B=1024，因为EDRAX矢量承诺支持的矢量大小是2的幂。

表1和表2显示了我们的微观基准测试的结果。我们没有在表1中提供EDRAX与SNARK的操作延迟，因为我们没有实现非常复杂的SNARK计算。我们在这里详细说明了这些成本。CommitUpdate保持不变。根据[17、§4.2]，Verify的成本大约是7ms。理解带有SNARK的ProofUpdate的成本更为微妙，因为EDRAX要求每当矢量中的任何数据发生变化时，都要更新*每个*矢量承诺证明（每个客户端一个），但只要求重新计算必须与交易一起发送的一个证明（发送方账户值）的SNARK。计算SNARK的成本是77s[36]，仅在提交交易时由客户产生；在所有其他时间，有SNARK的ProofUpdate与没有SNARK的ProofUpdate是一样的。

EDRAX的实现使用了与Aardvark不同的底层椭圆曲线。由于曲线操作在EDRAX曲线上的速度大约是5倍，所以CommitUpdate

特别是，Aardvark使用BLS12-381[6]椭圆曲线，而EDRAX使用两种BN曲线之一[7]（取决于SNARKs的存在）。然而，EDRAX实现中使用的特定曲线提供了大约100比特的安全性[5]，这比我们的曲线提供的128比特安全性要低。

为了比较独立于曲线选择的结构，我们在表3中统计了Aardvark和EDRAX的分组和配对操作的数量。(我们还注意到[47]的向量承诺与Aard-vark的表现相似）。我们的分析表明，在EDRAX的承诺中，Verify、Open和ProofUpdate比Aardvark的慢。

8.2 存储和带宽分析

节省存储。 Aardvark需要档案来存储数据库中的所有记录。如果s是键值对的数量，|k|是一个键的大小，|v|是一个值的大小，数据库的存档存储成本是

$$S_A = S(|k| + |v|)。$$

在没有Aardvark或其他认证的存储机制的情况下，这是每个验证者都会承担的存储成本。

和ProofUpdate操作也更快。然而，Verify时间（没有SNARK）大致相当。

[9] 最近的配对实现[42, 43]很可能为我们的基准提供一个明显的速度提升。

在Aardvark中，验证器必须每隔B条记录就向一个桶存储一个承诺。验证器还必须存储尾部桶中的记录，以及所有桶的所有记录在其缓存中。这个缓存最多有$2L$个桶大（至少有L个桶小，除非数据库小于L个桶大）。更多-过，验证器必须存储最后的τ个交易块，以处理不超过τ个块的情境。如果一个编码的交易块的大小是$|T|$，而一个向量承诺的大小是|c|，这意味着数据库的验证器存储成本为的上界如下：

$$S_V < \lceil s/B \rceil |c| + (2L + 1)(|k| + |v|)B + \tau|T|。$$

$2L$的一个自然选择是$\lfloor T \rfloor$的一个小倍数，其中$B|t|$ |t|是一个交易的大小，它允许验证者执行几个连续的删除请求块，而不考虑档案的可用性。如果$\lfloor T \rfloor = 10$，那么$2L = 30$的值就足够了。即使区块被确认为非常迅速（例如，一秒钟一次），设置τ = 1000允许客户相当多的并发性（例如，上下文保持着有效期为15分钟）。因此，由于上述等式中的最后两个项相当小--大约1GB为$|T| = 1$MB--我们可以在s增大的时候放弃它们。因此，验证器的成本和存档的成本之间的比率在边界是

$$\frac{S_V}{S_A} \to \frac{B(|k| + |v|)}{|c|}.$$

如第4节所述，在我们的特定提交方案中，|c| = 48字节。选择一个值，如B=1000的密钥的

规模|k|=32，规模|v|=8字节的值，使Aardvark验证器的节省系数超过800倍；我们注意到这个节省系数随着|v|的大小而增加。

带宽成本。交易的带宽成本包括传输交易的成本和传输交易密钥集中每个密钥k的值以及相应的上下文σ_k的成本。

无论是否存在一个密钥k，σ_k包括k本身、它被映射到的值v，钥匙的继承者 succ(k)，钥匙在顺序排序i中的索引，上下文的版本号τ_0，以及打开向量承诺π所产生的证明。此外，对于插入新k的上下文，交易还必须包含钥匙k本身，以及上下文的

最后，删除一个键k需要对整个桶进行解密；在摊销的情况下，一个净删除需要对一个槽进行解密（但没有证明）。如果交易数据的其余部分

(包含其操作)的长度为|t|字节，那么一个插入n个$_1$，修改n个$_2$，删除n个$_3$的交易所传输的字节数为

$$|t|+(3n_1 + 2n_2 + 3n_3)|k| \\ +(n_1 + n_2 + 2n_3)(|v| +|\pi| +|i| +|\tau_0|) \\ +max\{n_3 - n_1, 0\}(|k| +|v|)。$$

回顾一下，Aardvark的上下文是由一个版本号和一个字典槽排序的索引组成的，此外还有一个48字节的矢量承诺证明（见第4节）。

意味着大小为$|\pi| +|i| +|\tau_0| = 64$字节的上下文足以对应于一个特定的密钥。由于

首先传输交易的是

$$|t| +(n_1 + n_2)(|k| +|v|) + n_3 |k|。$$

我们得到，在|k| = 32和|v| = 8的情况下，每次投入操作的开销大约为100B，每次删除操作的开销为200B。

与交易规模相比，上下文开销很大，最适合于读写很多钥匙的小交易，而对于读写很少钥匙的大交易来说，开销最小。请注意，交易包括交易所读写的密钥集，这也是交易规模的一部分。

请注意，通过聚合证明，我们可以只传输一个单一的证明，这使边际开销减半，每个交易每个密钥大约增加50个字节，加上聚合证明的固定成本。交易访问许多不同密钥的应用从聚合中获益匪浅。

8.3 系统吞吐量

我们对Aardvark的计算开销进行了实验性评估。我们将Algorand加密货币[22]的存储后端替换为我们开源的Aardvark[29]，并对涉及Put或Delete操作的单一操作交易进行基准测试。

表4： 验证器处理100000笔交易的时间

场景	磁芯	平均值±SD(s)
放置(修改)	1	342 ± 14
放置（插入）	1	349 ± 13
删除	1	684 ± 38
放置(修改)	32	34 ± 1
放置（插入）	32	45 ± 2
删除	32	67 ± 3

(Get涉及与Put类似的处理)。我们分别测量存档和验证器的完成时间，以消除网络延迟的影响。

我们把Put操作分为两种情况：(1)插入新钥匙的Put操作和(2)修改字典中现有钥匙的Put操作；这使我们能够衡量更新钥匙继承者的额外成本。对于Delete操作，我们传入桶的预映射，因为它们需要用于向量解约。我们不对删除操作的证明进行汇总，而是分别传送两种证明。

我们产生的交易如下。键是32字节长，值是8字节长。对于插入密钥的Put事务，密钥是一个随机的32字节的字符串。对于其他
交易时，密钥从当前的密钥集中随机抽取。最大的交易寿命是τ=10个区块。交易的最小有效区块序列号是在b和b-τ之间均匀随机选择的。

字典以100万个随机键值对进行初始化，对应的总状态大小为47MB。[10]在运行每个工作负载之前，我们执行τ+2=12个块、
每个人都有10 000个相同类型的交易，以便
达到稳态行为。

我们在一个拥有Intel Xeon Platinum 3.0GHz CPU的`c5.metal`亚马逊EC2实例上进行实验，使用`numactl`来限制系统可用的物理CPU。我们从用Go编写的字典实现中调用矢量承诺方案的Rust实现。我们将承诺、最近的交易和（对于档案）槽存储在内存的SQLite数据库中，以减少测量中I/O延迟成本的影响。

验证者的吞吐量。我们预先生成100 000个交易，将它们均匀地划分为10个区块，然后将每个区块发给验证器。验证器并行地执行加密证明验证，并验证所有交易的证明--即使是那些只影响最近修改的密钥的交易，以模拟最坏情况下的性能。我们测量验证和应用每个区块中的交易的时间。我们运行十个

[10] 我们希望对计算开销的测量能够普及到更大的字典，因为加密成本构成了CPU的主要瓶颈。

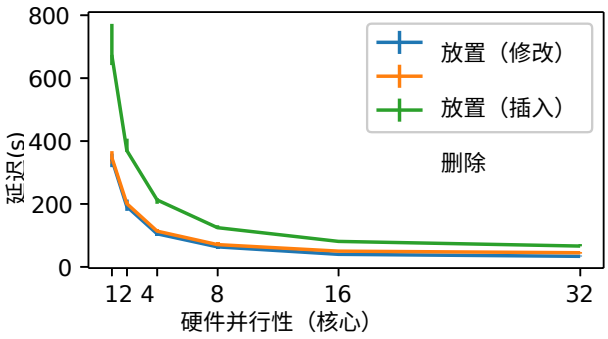


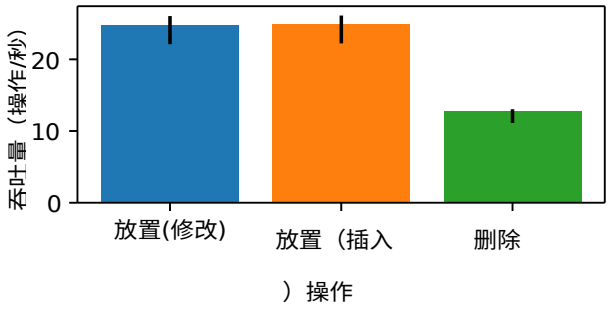
图3： Aardvark相对于核心数量的扩展。点代表了应用10个区块的100000个交易所需的中位时间，而误差条代表了最小值和最大值。增加核心可以提高证明验证的性能，证明验证是非常并行的，但更新承诺仍然必须按顺序进行。

在1、2、4、8、16和32个核心的情况下分别进行试验。我们在表4中总结了这些结果，并在图3中绘制了这些结果。

我们的实验表明，对于现有密钥来说，充满Delete交易的区块的交易验证时间大约是充满Put交易的区块的两倍，因为证明验证是昂贵的，Delete交易需要两次证明验证，而Put交易需要一次（除了验证尾槽）。

并行化提高了系统的吞吐量，但只是在一定程度上：从1个核心增加到32个核心提高了吞吐量
由于几个原因，在这一过程中，每一个交易都会影响到下一个交易。首先，在最坏的情况下，每个交易可能会影响到下一个交易，迫使序列化的
交易处理。第二，在每个区块中处理所有的delta会产生一个固定的成本，因为delta要与稳定存储进行协调。第三，尽管证明验证是完全并行的，但在我们的实验中，承诺更新是串行的（我们没有将承诺更新并行化）。尽管在普通情况下，一些并行是可能的，但对手可能会导致许多修改来影响一个桶的不同索引，在最坏的情况下迫使这些更新的串行化。[11]因此，当只有少数核心可用时，证明验证成本占主导地位，而随着并行化的增加，更新槽和承诺的开销成为瓶颈。

归档吞吐量。由于归档操作是可并行的，我们在单核上



评估归档工作负载。在每个试验中，我们查询一个单一的存档，为10 000个交易提供上下文。我们为每个工作负载运行10次试验，测量每次试验中创建上下文的吞吐量，并将结果绘制在图4中。

回顾一下，一个修改密钥的Put操作需要

[11] 批量承诺更新可能会改善这种最坏情况下的性能。原型中没有这项优化，留待今后的工作。

图4： 单个核心的单个存档上每秒创建的事务上下文。放置操作花费的时间大致相同，无论他们是插入一个新的密钥还是修改一个现有的密钥。删除操作需要同时执行Read和DeleteContext，所以产生一个证明需要两倍的时间。该图显示了吞吐量的中位数，误差条表示最大和最小值。

一个插入新键的操作需要查找其前任的上下文，而一个删除操作则需要每个查找中的一个。我们的评估显示，一个单核档案库可以提供大约12.8个Delete和大约2.5个Put操作。
每秒22.1个Put请求，不管是否插入了钥匙。这表明，Aardvark的成员资格证明和非成员资格证明所需的时间大致相同，因为它们是由创建加密证明的成本主导的，而不是其他成本。

9 结论

本文介绍了Aardvark，一个适合于高吞吐量、分布式应用的认证字典。我们表明，有可能通过基于配对的矢量承诺来创建具有简短的成员资格和非成员资格证明的认证字典，同时对额外的资源使用实施严格的限制。我们开发了一个版本管理方案，使我们能够完全忽略昂贵的证明更新成本，同时支持并发事务的执行。我们的评估表明，剩余的成本存在于证明验证和更新向量承诺中，并且部分地被并行化。

鸣谢

作者要感谢Hoeteck Wee和Adam Suhl对矢量承诺的分析和实施的帮助，感谢Alin Tomescu对论文的动机和安全模型的讨论。Yossi Gilad得到了希伯来大学网络安全研究中心、Alon奖学金和Mobileye的支持。本材料基于国家科学基金会研究生研究奖学金支持的工作，该奖学金编号为1745302。

参考文献

[1] Shashank Agrawal 和 Srinivasan Raghuraman.Kvac：区块链的键值承诺及其他。In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020*, pages 839-869, Cham, 2020.Springer Interna- tional Publishing.

[2] 阿尔戈兰。配对加库，2020年。https://github。com/algorand/pairing-plus。

[3] Algorand。Pointproofs的源代码，2020年。https：//github.com/algorand/pointproofs。

[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ra- mamurthy.Concerto：一个具有完整性的高并发键值存储。在Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *SIG- MOD 2017*, pages 251-266.ACM，2017。

[5] Razvan Barbulescu and Sylvain Duquesne.更新配对的密钥大小估计。*Journal of Cryptology*, 32:1298-1336, 2019.

[6] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott.构建具有规定嵌入度的椭圆曲线。In Stelvio Cimato, Giuseppe Persiano, and Clemente Galdi, editors, *Security in Communica- tion Networks*, pages 257-267, Berlin, Heidelberg, 2003.Springer Berlin Heidelberg.

[7] Paulo S. L. M. Barreto 和 Michael Naehrig.质数的配对友好椭圆曲线。In Bart Pre- neel and Stafford Tavares, editors, *Selected Areas in Cryptography*, pages 319-331, Berlin, Heidelberg, 2006.Springer Berlin Heidelberg.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza.Zerocash：来自比特币的去中心化的匿名支付。In *SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 459-474.IEEE计算机协会，2014年。

[9] Alex Biryukov, Daniel Feher, and Giuseppe Vitto.zcash中的特权方面和潜意识渠道。In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS 2019, London, UK, November 11-15, 2019*, pages 1795-1811.ACM，2019年。

[10] Manuel Blum, William S. Evans, Peter Gemmell, Sam- path Kannan, and Moni Naor.Checking the correct- ness of memories.In *FOCS 1991, San Juan, Puerto Rico, 1-4 October 1991*, pages 90-99.IEEE计算机协会，1991年。后来作为[11]出现，可在http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991上查阅。

[11] Manuel Blum, William S. Evans, Peter Gemmell, Sam- path Kannan, and Moni Naor.检查记忆的正确性。*Algorithmica*, 12(2/3):225-244, 1994.可在 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991。

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch.累积器的批处理技术，应用于iops和无状态区块链。In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, vol- ume 11692 of *LNCS*, pages 561-586.Springer, 2019.

[13] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro.Coda：大规模的去中心化加密货币。*IACR Cryptol. ePrint Arch.*, 2020:352, 2020.

[14] Sean Bowe。配对，2019。https://github.com/zkcrypto/pairing。

[15] Vitalik Buterin. 无状态客户端概念，2017年。https://ethresear.ch/t/the-stateless-client-concept/172。

[16] Vitalik Buterin.一个不完整的卷积指南，2021年。https://vitalik.ca/general/2021/01/05/rollup.html。

[17] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang.Edrax：一种具有无状态交易验证的加密货币。Cryp- tology ePrint Archive, Report 2018/968, 2018.

[18] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Pra- teek Saxena, Elaine Shi, and Emin Gün. 论去中心化区块链的扩展。In *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[19] Scott A. Crosby and Dan S. Wallach.超高效地聚集与历史无关的持久性认证字典。In Michael Backes and Peng Ning, editors, *ESORICS 2009*, volume 5789 of *LNCS*, pages 671-688. Springer, 2009.

[20] 贾斯汀-德雷克。累积器、UTXO区块链的可扩展性和数据可用性.https://ethresear.ch/t/accumulators-scalability-of- utxo-blockchains-and-data-availability/176,2017.

[21] 萨迪厄斯-德瑞亚。Utreexo：为比特币UTXO集优化的基于哈希值的动态累加器。*IACR Cryptol. ePrint Arch.*, 2019:611, 2019.

[22] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich.Algorand：为加密货币扩展拜占庭协议。In *SOSP 2017, Shanghai, China, October 28-31, 2017*, pages 51-68. ACM, 2017.

[23] Alex. IntroducingzkSync: the missinglink tomass adoption of Ethereum. adopted of Ethereum. https://medium.com/matter-labs/introducing-zk-sync-themissing-link to-mas-adoption-of-ethereum-14c9cea83f58.

[24] Michael T. Goodrich, Michael Shin, Roberto Tamassia, and William H. Winsborough.新属性凭证的认证字典。In Paddy Nixon and Sotirios Terzis, editors, *Trust Management, First International Conference, iTrust 2003, Heraklion, Crete, Greece, May 28-30, 2002, Proceedings*, volume 2692 of *LNCS*, pages 332-347.Springer, 2003. 可在 http://cs.brown.edu/cgc/stms/papers/itrust2003.pdf。

[25] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang.Pointproofs：多个矢量承诺的聚合证明。*IACR Cryptol. ePrint Arch.*, 2020:419, 2020.

[26] Jens Groth.论基于配对的非交互式论证的大小。In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305-326.Springer, 2016.

[27] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. 对多项式的恒定尺寸承诺及其应用。In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177-194.Springer、2010.

[28] Jonathan Lee, Kirill Nikitin, and Srinath T. V. Setty.无复制执行的复制状态机。In *SP 2020*, pages 119-134.IEEE，2020年。

[29] Derek Leung, Leonid Reyzin, and Nickolai Zeldovich.Aardvark 原型文物，2020 年。https://github 。 com/derbear/aardvark-prototype。

[30] Benoît Libert and Moti Yung.简明的水星向量承诺和独立的零知识集与短证明。在Daniele Micciancio，编辑，*TCC 2010*，*LNCS*第5978卷，第499-517页。Springer、2010.

[31] Loopring: zkRollup 交换和支付协议。https://loopring.org/。

[32] Ralph C. Merkle.一个认证的数字签名。In Gilles Brassard, editor, *CRYPTO '89*, volume 435 of *LNCS*, pages 218-238.Springer, 1989. 可在 http://www.merkle.com/papers/Certified1979.pdf。

[33] Andrew Miller. 在平衡的Merkle树中存储UTXO（零信任节点与O(1)- 存储），2012 年。https://bitcointalk.org/index.php?topic=101734.msg1117428。

[34] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi.认证的数据结构，一般情况下。在Suresh Jagannathan和Peter Sewell，编辑，*POPL '14*，411-424页。ACM，2014。项目页面和完整版本见 http://amiller.github.io/lambda-auth/ paper.html。

[35] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh.使用高效集合累加器扩展可验证的计算。在Srdjan Capkun和Franziska Roes- ner，编辑，*USENIX安全2020，2020年8月12日至14日*、第2075-2092页。USENIX协会, 2020.

[36] Babis Papamanthou.私人通信。

[37] Charalampos Papamanthou and Roberto Tamassia.两方认证数据结构的时间和空间效率算法。 In Sihan Qing, Hideki Imai, and Guilin Wang, editors, *ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceed- ings*, volume 4861 of *LNCS*, pages 1-15.Springer, 2007. 可在 http://www.ece.umd.edu/~cpap/ published/cpap-rt-07.pdf。

[38] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos.动态集上操作的最佳验证。在Phillip Rogaway的编辑下，*CRYPTO 2011*，*LNCS*第6841卷，第91-110页。Springer、2011.

[39] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos.基于加密累积器的认证哈希表。*Algorithmica*，74（2）：664-712，2016。

[40] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov.改进认证的动态字典，并应用于加密货币。In Aggelos Kiayias, editor, *FC 2017*, volume 10322 of *LNCS*, pages 376-392.Springer, 2017.

[41] Y. Sakemi, T. Kobayashi, T. 斋藤，和R.Wahby. 配对友好曲线。 IETF草案、https://tools.ietf.org/id/draft-irtf- cfrg-pairing-friendly-curves-07.xml# 基于配对的加密技术的应用。

[42] SCIPR-Lab.Zexe, 2020. https://github.com/scipr-lab/zexe.

[43] Supranational. blst, 2020. https://github.com/supranational/blst.

[44] Roberto Tamassia and Nikos Triandopoulos.数据结构的认证和验证。In Al- berto H. F. Laender and Laks V. S. Lakshmanan, editors, *AMW 2010*, volume 619 of *CEUR Workshop Proceed- ings*.CEUR-WS.org, 2010.

[45] Peter Todd. Making UTXO set growth irrele- vant with low latency delayed TXO commitments, 2016. https://petertodd.org/2016/delayed-txo-commitments.

[46] Peter Todd, Gregory Maxwell, and Oleg Andreev.Reducing utxo: users send parent transactions with their merkle branches. https://bitcointalk.org/index.php? topic=314467, 2013.

[47] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich.无状态加密的可聚合子矢量承诺。In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *LNCS*, pages 45-64.Springer, 2020.

[48] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas.通过仅附加认证的字典实现透明度日志。In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *CCS 2019*, pages 1299-1316.ACM，2019年。

[49] Alin Tomescu, Yu Xia, and Zachary Newman.具有交叉递增证明（不）聚合的Au-thenticated字典。*IACR Cryptol. ePrint Arch.*, 2020:1239, 2020.

[50] 比尔-怀特。轻量级加密货币分类账的理论。可在https://github.com/bitemyapp/ledgertheory/blob/master/lightcrypto.pdf（见也见代码at https://github.com/bitemyapp/ledgertheory), 2015.

[51] Gavin Wood等人，Ethereum：一个安全的去中心化的通用交易账本。*以太坊项目黄皮书*，151（2014）：1-32，2014。

[52] Yupeng Zhang. vector commitment scheme with efficient updates, 2019. https://github.com/starzyp/vcs.

A 安全分析

Agrawal和Raghuraman[1]定义了键值承诺的安全性，其功能与我们的认证字典非常相似。为了定义安全性，我们首先生成了他们的密钥绑定概念[1，§3.2]，然后将这个概括应用于Aardvark。

Aardvark的操作是Get、Put和Delete，与Agrawal和Raghuraman的操作不同，后者是Insert和Update。而不是采用零散的方法，简单地将这些操作替换到认证的字典定义中、

我们遵循[38, 39, 44]的方法，为*任何*一组操作定义了一个通用的认证数据结构，其中包括不同的可能的字典接口，以及其他具有完全不同接口的数据结构（例如，范围查询，按等级搜索）。在定义了通用数据结构后，我们将其专用于Aardvark，使用地图而不是Agrawal和Raghuraman的图元集，并以地图更新的方式定义了Put和Delete。

A.1 一般认证的数据结构定义

让F：$S\times O\times X\to S\times Y$表示一个*理想的*数据结构功能。("理想 "指的是数据结构在没有任何对抗性的，甚至是意外的偏差的情况下，如预期的那样拥有。)这里S是一些数据结构的*理想状态*的集合，有一些初始状态$s_{\text{init}} \in S$，它支持一组
由一组*操作代码*（或*操作代码*）确定的*操作。*
代码）O.给定一个状态和来自集合X的输入，一个操作产生一个新的状态和集合Y中的一个输出。

现在我们转向实现F的认证（"真实"，而不是 "理想"）数据结构。让$\Delta : \mathbb{N} \to \mathbb{N}$是一个*延迟函数参数*（它将决定确切的
完备性的定义）。回顾一下，一个认证的数据结构涉及一个验证者和核查者。验证者（例如Aardvark档案馆）持有一个从某个集合W中抽取的大的*真实状态*，而核查者（例如Aardvark验证者）持有一个
$_{\Delta}$验证者的目标是在对数据结构进行读写的情况下保持其承诺$c\in C_{\Delta}$，验证者通过从一些*上下文*集Σ中产生一个*上下文*σ[12]使验证者能够做到这一点。

我们希望对一个应用程序（例如，加密货币）进行建模，其中验证者产生操作上下文的顺序可能与验证者执行操作的顺序不同。特别是，由于操作的某些部分将是固定的（例如，字典中的键），而某些部分将是可变的（例如，键在某个时间点被映射到的值），我们将输入X分解为一个*静态*部分$_{\text{Xstatic}}$和一个
*动态*成分$_{\text{Xdynamic}}$（即，$X = {}_{\text{Xstatic}} \times {}_{\text{Xdynamic}}$）。对于
验证者，定义两个函数。$P_{\Delta} : W_{\Delta} \times O \times \text{Xstatic} \to$ Σ是一个*证明生成函数*，它消耗一个真实状态、操作码和静态输入，并产生一个操作的*上下文*。$G : W \times O \times X \to W$是相应的*实数更新函数*，它可以改变实数状态。

让Init是某种概率算法，它的运行时间与*安全参数* λ∈N成多项式关系，这样Init(λ)=(c_{init}，w_{init}，pp)，其中$c_{\text{init}} \in C_{\Delta}$是*初始承诺*，$w_{\text{init}} \in W$是*初始状态*，pp是认证数据结构的公共*参数*。

[12] 上下文提供了计算操作结果和更新承诺的所有必要信息。它们尤其包括传统上被认为是证明的信息。

定义$V_{pp} : C_\Delta \times O \times {}_{Xstatic} \times \Sigma \to \{0, 1\}$是一种*验证*算法，它将一个（承诺、操作码、输入、上下文）元组映射到一个位，表示上下文是否有效、
给定一些固定的初始参数。

定义*承诺更新函数*$F_{V,pp} : C_\Delta \times O \times X \times \Sigma \to C_\Delta \times Y$是一种算法，它将（承诺、操作码、输入、上下文）图元映射为（承诺、输出）对。

A.1.1 完整性

1. 让$(c_{init}, w_{init}, pp) \leftarrow Init(\lambda)$
2. 设$s := s_{init}$，$c := c_{init}$，$w := w_{init}$，$i := 0$。
3. 无限次重复：
   (a) 设置$w_i := w$。
   (b) 任意选取$o \in O$，$x = (x_s, x_d) \in X$，$j := \{0, 1, \ldots, \Delta(i)\}$非确定地。设$\sigma := P(w_{i-j}, o, x_s)$。
   (c) 如果$V_{pp}(c, o, x_s, \sigma) = 0$，输出FAIL并停止。
   (d) 设$(s, y) := F(s, o, x)$，$(c, y^-) = F(c, o, x, \sigma)$，$w := G(w, o, x)$，and $i := i + 1$

如果对于上述算法中的所有非确定性选择，算法从未输出FAIL，我们就说F对于某个Δ满足Δ-同步*完备性*。很自然地，*同步完备性*对应于$\Delta(n) = 0$，*异步完备性*对应于$\Delta(n) = n$。

A.1.2 健全性

定义A是一个*对手*，它是一些概率性的进程，其运行时间为λ的多项式。将$G_{F,\lambda,A}$定义为是由以下算法返回的值。

1. 让$(c_{init}, w_{init}, pp) \leftarrow Init(\lambda)$
2. 设$c := c_{init}$，$s := s_{init}$。
3. 对于一些与λ成多项式的迭代次数，在一个循环中做以下工作：
   (a) 查询A，得到$(o, x, \sigma) \leftarrow A(pp, \lambda)$ 其中$o \in O$，$x = (x_s, x_d) \in X$，$\sigma \in \Sigma$。
   (b) 设$(s', y) := F(s, o, x)$　　并设$(c', y^-) = F(c, o, x, \sigma)$。
   (c) 如果$V_{pp}(c, o, x_s, \sigma) = 1$，并且$y \neq y^-$，则输出FAIL并停止。
   (d) 如果$V_{pp}(c, o, x_s, \sigma) = 1$，设$c := c'$，$s := s'$。
4. 输出OK并停止。

我们说F是F的*安全*实现，如果对于所有这样的A，$\Pr[G_{F,\lambda,A} = \text{FAIL}]$在$\lambda$中是可以忽略的。

请注意，我们的稳健性概念假定c_{init}是基因的。在另一种情况下，A生成c，这个概念要求A不能使验证者接受两个结果相反的操作。在另一种情况下，即A生成c_{init}，这个概念将要求A不能使验证器接受两个结果相互矛盾的操作。我们的应用不考虑对抗性生成的com- mitments，因为验证器知道正确的c_{init}，并在每次操作后自己更新这个承诺。

A.2 理想字典的定义

现在，我们把我们的一般定义专门化为一本字典。

让K是一个键的集合，V是一个值的集合。让$\perp$是一个哨兵值，表示不存在$\perp \notin K$和$\perp \notin V$。记为$V_\perp = V \cup \{\}$。

定义$M : K \to V_\perp$是K到$V_\perp$的*字典映射*。让$S = M$。定义*空字典*为$m_{empty} \in M$，其中$m(k) = \perp$为所有$k \in K$。

称读为*读操作码*，写为*写操作码*。让O={读，写}。对Get(k)的调用对应于功能$F(m,$ read$, (k,))$。请注意，由于$\perp$表示不存在，对于任何m和$v = \perp$对应于功能F（m，写，（k，v））；同样地，Delete（k）对应于F（m，写，（k，））。

在理想状态$m \in M$上定义F如下，opcode $o \in O$，而输入（k，v）$\in K \times V_\perp$。

- F（m，读，（k，）） =（m，m（k））。
- F（m，写，（k，v））=（m'，），其中$m'(k) = v$，并且
  $m'(k') = m(k')$ 对于所有$k' \neq k$。

在这种情况下，$_{Xstatic} = K$，$_{Xdynamic} = Y = V_\perp$。

为了简化我们的分析，我们分析了Aard-的安全性。vark是关于操作的，而不是关于交易的。给定一个带有验证算法V和更新函数G的安全操作字典，我们可以定义一个相应的s-治愈交易字典，有一个验证算法V'和更新函数G'，其中G'是G对交易中每个操作的组合，V'返回1，如果且仅是如果V对组成中的每个中间状态返回1。

A.3 土豚的安全

定理1. *让F由Aardvark实现，C_Δ是验证器状态集，Σ是相应的操作上下文，*Init*是创建初始vec- tor承诺参数的函数（§4），$c_{\text{init}} = \text{Commit}(m_{\text{empty}})$。那么理想字典的Aardvark实现就满足了健全性和τ-同步完整性。*

完备性由第6节中的论证得出。为了证明健全性的成立，考虑Aardvark对理想数据结构m的表述，我们称之为a*快照*。虽然m只是简单地将键映射到值，但快照将这个映射表示为一个槽的向量e^{-}，每个槽都包含一个键、一个值和后续的键。我们认为每次操作后都有两个不变量：当前的快照忠实地代表了当前的m，而验证者对过去的τ个快照持有承诺。

这些不变量足以表明健全性的成立。事实上，为了获胜，A必须输出一些$(o, (k, v), \sigma)$，使得$V_{\text{pp}}(c, o, k, \sigma)=1$和$F(c, o, (k, v), \sigma)$输出一些与$F$不一致的$y^{-}$，意味着$o$=读和$y'=m(k)$。

上下文σ必须是一个不超过τ个操作的快照e^{-}。根据vec- tor承诺的约束属性，σ将包含e^{-}的k的正确时隙信息，否则验证器（由于验证器根据第二个不变量对$e^{-'}$有正确的承诺，所以它得到了正确的输入）将以所有但可忽略不计的概率拒绝。[13]因为验证器存储了过去的τ操作，所以验证器可以根据当前的快照正确计算k的槽信息，根据第一个不变式，可以得到m（k）。这与$y' = m(k)$相矛盾。

为了论证不变量的成立，我们需要推理一下操作如何改变M、快照和验证器的状态。在第A.3.2节中，我们证明在每个操作之后，验证器继续持有对过去τ个快照的承诺。直观地说，向量承诺的安全性确保了上下文σ对应于正确的槽信息，并且
因此，向量承诺将被更新的价值----。
忠实地提交到新的快照的idator。

在第A.3.3节中，我们根据数据结构设计论证了快照在每次操作后继续忠实地代表理想地图m。特别是，我们展示了一个函数的存在，该函数将每个快照映射到一个单一的m，并且尊重由写执行的更新。

A.3.1 建立验证器行为的模型

我们首先更精确地描述了Aardvark的（抽象）行为。

让$e_i = (k_i, v_i, k')$对于一些$k_i, k' \in K, v_i \in V$是一个槽，$e^{-} = e_0, \ldots, e_{S-1}$是快照。我们假设初始快照由一个槽$(\perp, \perp, \in)$组成，其中$\perp$是一个哨兵密钥，它比每个密钥都大。

我们把对一个快照的*承诺定义*为列表$c^{-} = c_0, \ldots, c_{\lceil S/B \rceil - 1}$，其中$c_j$ = Commit($e_{jB}, \ldots, e_{\min((j+1)B-1, S)}$）（最后一个向量用零槽填充到长度$B$）。

我们说Aardvark*操作*是一个元组γ=（o，（k，v），σ）。对于任何操作γ，我们定义函数$A(e^{-}, \gamma) = e^{-'}$，它将操作γ*应用*于快照$e^{-}$，并产生一个新的快照$e^{-'}$，方法与Aardvark（§5）的定义相同。

A.3.2 验证器被绑定到快照上

推理1. *在t次操作之后，验证者持有对过去*min(t, τ)*快照的承诺，以及*min(t, τ)*最近的操作，其概率都是可以忽略不计的。*

*证明。*我们通过对t的归纳法来证明这条定理。

基本情况在设计上是成立的，所以考虑归纳假设。对一个密钥k的任何操作γ必须有一个相对于快照$e^{-}_{t'}$的上下文，其中$t - \tau \le t' \le t$，否则的话

[13] 形式上，如果A可以导致验证器接受不准确的信息，那么A可以通过从头运行交易，在多项式时间内违反向量承诺的约束属性。

的验证器将被拒绝。根据向量承诺的约束属性，上下文将包含关于快照e^{-}_{t}′的持有k的槽的准确信息（在删除的情况下是其前身），否则向量承诺验证算法（通过归纳假设得到正确的输入，因为验证器有正确的c^{-}_{t}′）将以所有但可忽略的概率拒绝。如果验证器拒绝该操作，那么快照和验证器的状态都不会改变，归纳法的情况就成立。

如果验证者接受该操作，它将使用CommitUpdate来计算快照e^{-}_{t+1}的新承诺。因为验证器有相关的信息对来自e^{-}_{t}′的操作和所有发生的操作
自t'，验证器可以正确地计算最新的快照e^{-}_{t}和由操作产生的新快照e^{-}_{t+1}之间的差异。根据CommitUpdate的正确性、
验证器将计算出对这个新快照的正确承诺，并且归纳的情况成立。

A.3.3 快照编码的理想地图

我们必须证明的第二个不变性是，快照对理想地图的编码方式与读和写操作一致。由于读既不改变快照也不改变地图，我们只关注写。如果以下条件成立，我们定义快照e^{-}为*格式良好*。

- 对于任何一个k，一个三联体（k, v_1, succ(k)）在e^{-}中最多出现一次。
- 如果$v \neq \perp$，并且（k，v，k'）$\in e^{-}$，$v \neq \perp$。换句话说，所有在槽中的键（不是哨兵键$\perp$）都被映射到一个不是$\perp$的值。
- 对于所有(k, v, k')$\in e^{-}$，情况是：k' = succ(k) (rela-)对e'中的键列表而言是有意义的）。换句话说，继承者-e^{-}中的引用形成一个有效的循环链接的键列表。

从这个定义可以看出，存在一个自然的*解码*函数D，从格式良好的快照到理想地图m。也就是说，因为对于所有的槽（k，v，k'）$\in e^{-}$，（k，v）出现了
唯一的，由此可见，D定义了地图m，其中$m(k) = v$如果(k, v)出现在e^{-}中，$m(k) = \perp$，否则。

现在我们将通过归纳法来证明该不变性的成立。通过我们的初始化，基本情况就出来了。归纳步骤是由以下定理给出的。

推理2。 *假设* e^{-} *是格式良好的，*D(e^{-}) = m，γ = (写，(k, v), σ)。*让*e^{-} = A(e^{-}, γ)。*让* m' *是由* $m'(k) = v$ *和* $m'(k') = m(k')$ *为所有* $k' \neq k$ *而给出的映射。那么* $e^{-'}$ *是格式良好的，并且* D(e^{-}) = m'。

*证明。*通过对A的检查可以得出该结论（对于$v = \perp$和$v' = \perp$的情况），定义见A.3.1节。 □

□