Symphony：通过协调实现的表达性安全多方计算

Ian Sweet[a]、David Darais[b]、David Heath[c]、William Harris[b]、Ryan Estes[d] 和 Michael Hicks[a, e]

a 美国马里兰大学 b 美国 Galois 公司
c 美国佐治亚理工学院 d 美国佛蒙特大学 e 美国亚马逊

摘要

上下文 安全多方计算（MPC）是指一类密码学技术：相互不信任的参与方可以利用各自的私有输入计算函数，同时仅揭示函数输出。**问题** 使用现有的编程语言和框架，往往难以正确且高效地编写多方计算程序，尤其是在需要协调不同计算角色时。如何让这一过程变得更加容易？

方法 我们提出了 Symphony，这是一种用于两个或更多参与方之间多方计算的新型函数式编程语言。Symphony 从先前多方计算语言所采用的单指令多数据（SIMD）语义出发，在这种语义中，每个参与方承担对称的职责；随后，它通过引入可协调多个参与方的构造，将这一语义加以泛化。Symphony 引入了 一等份额 和 一等参与方集合，以提供无与伦比的语言级表达能力，同时保持高效率。

知识 通过开发一种核心形式语言 λ-Symphony，我们证明了程序直观的广义 SIMD 视图与其实际的分布式语义一致。因此，程序员可以自上而下地阅读程序并对其进行推理，尽管实际上程序是在多台机器上以协调方式分布运行的。我们利用多个密码学后端实现了 Symphony 的原型解释器。借助该解释器，我们编写了各种多方计算程序，发现 Symphony 能够表达其他语言无法表达的优化协议，而且 Symphony 程序通常都能高效运行。

依据 除了开发形式化证明、原型实现和多方计算程序案例研究之外，我们还测量了 Symphony 实现运行多个基准程序时的性能，发现运行相同程序时，其性能与 Obliv-C 相当。Obliv-C 是面向 C 语言的先进两方多方计算框架。我们还测量了 Symphony 在一种优化的安全洗牌协议上的性能。该协议基于一种以往的语言都无法表达的协调模式，测得的性能远优于标准替代方案。

重要性 对多方计算的编程需求日益增长，各种语言和框架也层出不穷。本研究降低了编程门槛，让程序员能够编写高效、协调的多方计算程序，并对其进行推理和理解。本研究适用于希望设计新应用和协议的开发者与密码学家，他们可以在语言层面进行设计，而不必处理底层密码学细节。Symphony 的λ-Symphony形式化及相关证明也出奇地简单，可作为后续开展多方计算和分布式编程形式化研究的基础。所有代码和研究材料均已开源。

ACM CCS 2012

- **安全与隐私 → 逻辑与验证;**

关键词 安全多方计算、协调、编程语言、形式元理论、分布式编程

《编程的艺术、科学与工程》

提交日期 2023年1月1日

发布于 2023年2月15日

DOI 10.22152/programming-journal.org/2023/7/14



© Ian Sweet、David Darais、David Heath、William Harris、Ryan Estes 和 Michael Hicks 本作品根据“CC BY 4.0”许可协议授权。发表于《编程的艺术、科学与工程》第 7 卷第 3 期，2023 年，第 14 篇文章；共 55 页。

1 引言

安全多方计算（MPC）是密码学的一个分支，允许彼此互不信任的参与方对各自的私有输入执行任意函数，同时只揭示函数输出。也就是说，MPC 允许参与方在加密状态下运行程序。如今的安全多方计算技术比仅仅 20 年前的技术快数亿倍，这意味着现在能够完成的计算大大增多。现代框架也越来越便于使用，为程序员提供熟悉的语言来编写其 MPC 程序 [36, 2, 14, 39, 26, 32, 5]。

遗憾的是，尽管这些框架允许程序员表达许多多方计算，但当面对需要非平凡协调的问题时，它们仍显得力不从心。绝大多数安全多方计算框架默认认为，所有参与方都以单指令多数据（SIMD）的方式执行相同的同步活动。这种简化的视图并不总是恰当的，在许多场景中，让参与方执行不同的计算会很有用。例如，假设我们希望实现一款回合制纸牌游戏，其中N 名玩家使用多方计算共同洗牌和发牌。通过使用多方计算，各参与方可以确保牌堆和玩家各自的手牌保持机密。同时，每个参与方都可以根据自己的策略选择出牌，因此各参与方可能执行不同的操作，例如通过输入/输出与用户交互。此外，一旦某个参与方被淘汰出局，它还可能完全退出计算。再举一个例子，假设有大量参与方希望为一项隐私保护计算提供输入。在这种情况下，让参与方选出一个小型委员会代表他们执行计算是很实际的做法。这样做可以大幅提高效率，因为大多数安全多方计算基本构件的性能会随着参与方数量的增加而下降。

大多数现有的 MPC 框架都不提供协调功能；对于非同步情况，要么采用临时机制处理，要么完全不予处理。临时机制可能导致编程错误，进而造成（可能不确定的）挂起或错误答案。语言 Wysteria [28]提供了协调支持，但缺乏 表达能力 和易用性。例如，单个参与方无法将计算委托给其他参与方，而且 MPC 必须用一种僵化的子语言表达，使加密计算与明文计算交错执行变得不必要地繁琐。

我们的方法 我们提出了 Symphony，这是一种表达能力强的 MPC 语言，提供了适当的工具来协调涉及大量参与方的 MPC。Symphony 最有趣的语言特性如下：

- 作用域并行表达式块（即*par* 块）让程序员能够轻松控制哪些参与方执行哪些代码。程序员为*par* 块标注应进入该块的参与方集合。语言确保在块内执行的参与方对局部变量的逻辑值达成一致，因此不会出现死锁或未定义行为。重要的是，按照编舞式编程语言的风格 [8, 9, 10, 25], ，我们证明开发者可以像程序以单线程方式运行一样推理程序。

- 参与方集合是语言中的一等值：它们可以在运行时组合和计算，并用于标注*par*块，从而支持表达能力极强、可动态确定的协调操作。一等份额以一种允许程序员自由委托计算、将加密值从一个参与方集合重新共享到另一个参与方集合，并
- 响应式地混合多方计算操作与明文操作的方式，对多方加密值进行建模。

这些特性此前已部分出现在其他多方计算框架中：并行代码块和参与方集合的灵感来自 Wysteria，而一等份额则借鉴了 EMP [36] 和 Obliv-C [39]的编程风格。Symphony 是首个谨慎地结合 这些特性的多方计算语言，并对其进行了泛化，例如允许重新共享，并将已加密的值委托给不同的参与方集合，还能自由构造参与方集合并在其上进行计算，以细化协调操作。这些特性的组合对于编写此类协议至关重要：参与方可以自由加入、离开，并变更密码计算的执行位置。

贡献 我们的具体贡献如下。

- 我们阐述 Symphony 的必要性，通过示例介绍其关键特性，并将其与当前最先进的方案进行比较（第 2 节和第 3 节）。我们提出了 Symphony 的一种核心形式体
- 系，称为 λ-Symphony——包括语法（第 4.1 节）、单线程 语义（第 4.2 节）和分布式 语义（第 5 节）。我们证明，单线程语义能够忠实地表示分布式语义，因此可作为推理 Symphony 程序行为的基础（第 6 节）。我们介绍了 Sy
- mphony 的原型实现，该实现采用 EMP [36]和 MOTION [5]作为密码学后端（第 7 节）。我们讨论了用 Symphony 实现的 16 个程序，展示了与以往语言相比，它在易用性和表达能力 方面的优势（第 8.1 节）。我们还表明，尽管
- Symphony 表达能力强，但仍具有良好的性能。在模拟局域网上运行的一组内核基准测试中，Symphony 的耗时均值为 Obliv-C 的 15× 倍（第 8.2 节）。
-

Symphony 可在以下网址公开获取：https://github.com/plum-umd/symphony-lang。

2 背景与相关工作

本节介绍相关多方计算编程框架的一些背景，并详细探讨协调问题。

2.1 先前框架概述

表 1 根据底层多方计算协议的特性（第 2–5 列）以及计算的编程方式（第 6–7 列），概述了现有的主流 MPC 框架。作为一个代表性示例，下面给出 Obliv-C 中的“百万富翁问题”：两个参与方 A 和 B 希望在不透露各自总财富的情况下得知谁更富有：

表 1 MPC 框架比较。**协议** 表示底层密码机制使用的是混淆电路（GC）、线性秘密共享（LSS），还是同时利用二者的自定义混合（Hy）协议；# 表示参与安全计算的参与方数量；**TM** 表示协议所应对的威胁模型，即半诚实（S）、恶意（M）或二者兼有；**域** 表示该框架支持基于布尔值（B）电路、算术运算（A）电路，还是二者皆支持的 MPC；**类型** 表示该 MPC 框架是自定义语言（DSL）、语言扩展（Ext）还是库（Lib）；**特性** 表示该框架对反应式 计算（R）、从输入输出方到计算方的委托（D），以及多个参与方之间的同步随机性（$）的支持程度。

	原型	#	TM	Dom	Type	特性 R	D	$
EMP-toolkit [36]	GC	2	S,M	B	库 (C++)	●	○	◖
Obliv-C [39]	GC	2	S	B	扩展（C）	●	○	●
ObliVM [24]	GC	2	S	B	DSL	●	○	●
TinyGarble [33]	GC	2	S	B	扩展（Verilog）	○	○	○
Wysteria [28]	LSS	2+	S	B	DSL	●	○	◖
ABY [12]	GC,LSS	2	S	A,B	库 (C++)	○	○	◖
MOTION [5]	GC,LSS	2+	S	A,B	库 (C++)	○	○	◖
SCALE-MAMBA [2]	Hy	2+	S,M	A	DSL	●	○	●
Sharemind [27]	Hy	3	S	A	DSL	●	●	○
MPyC [32]	LSS	3+	S	A	库 (Python)	●	○	●
PICCO [40]	Hy	3+	S	A	扩展（C）	○	●	○
Frigate [26]	GC	2	S,M	B	DSL	◖	○	◖
CBMC-GC [14]	GC	2	S,M	B	扩展（C）	○	○	○
Viaduct [1]	GC,LSS	2	S	A,B	DSL	○	○	◖
Symphony	混淆电路、LSS	2+	S	A,B	DSL	●	●	●

```
1 voidmillionaire(void* args) {2 协议IO*io =
ar
my
my
re
```

双方以 SIMD 风格运行此程序。变量 `v1` 和`v2` 根据 `obliv` 关键字进行加密。函数 `feedOblivInt` 设置这些变量的初始值。在参与方 A 中，第 3 行对存储在 A 的 `io->mywealth` 副本中的输入进行加密，而第 4 行对参与方 B 执行同样的操作。在内部，该函数会将其加密输入发送给另一参与方，而另一参与方会同步接收该输入；也就是说，第 3 行从 A 发送到 B，第 4 行从 B 发送到 A。第 5 行在双方对这些加密值进行计算，计算结果本身也是加密的。最后，`revealOblivBool` 在双方之间进行协调，以解密结果；结果存储在每个参与方的`&io->cmp`中。

在底层，这些框架使用不同的多方计算协议（例如，两方混淆电路 [37], *N*方线性秘密份额 [16, 4], 或混合 [11] 协议），并提供

各种基本运算（布尔运算、算术运算 或两者兼有），并考虑不同类型的对手（遵循协议的“半诚实”对手，或破坏协议的“恶意”对手）。

有时，受底层特性的影响，其上层编程特性也有所不同；表 1 列出了三项：反应式多方计算 （R）、委托 （D）和同步随机性 （$）。反应式多方计算是指多次调用底层多方计算协议，后续调用可以利用先前调用所揭示的结果。例如，Obliv-C 支持反应式多方计算，因此我们可以扩展上面的百万富翁示例，在后续计算中使用已揭示的`&io->cmp`，并结合`A` 和`B`提供的其他输入。反应式多方计算对于优化（例如，联合中位数 [22] 和私有集合求交 [20]）以及交互很有用（例如，牌组经过加密的多轮纸牌游戏）。

委托 是指能够指定输入输出方，由其提供输入并接收输出，但不直接参与安全计算。委托对于可扩展性非常重要——一个在 100 个参与方上运行难以实现的多方计算协议，通过委托给一个由 3 人组成的委员会来运行，可能就变得可行。现代多方计算协议通常提供委托所需的密码学支持（通过对输入和输出进行加法共享），但缺少协调非计算方所需的编程支持；这与基本的 SIMD模型并不匹配（见第 2.2 节）。

最后，同步随机性 指多个参与方能够高效访问同一串随机数；这与委托结合使用，可以极大地帮助优化反应式多方计算 [19]。许多不支持此功能的语言可以扩展以实现它，在表格中用半圆标示。至于其他语言：CBMC-GC 和 TinyGarble 会编译为标准布尔电路，而标准布尔电路没有用于均匀比特生成的门；PICCO 和 Sharemind 基于阈值多方计算协议，只要求声明的参与方中的一个子集执行，而要在动态子集中高效地实现同步并非易事。

2.2 协调

SIMD 计算模型意味着，最容易指定的是执行相同操作的参与方。但如果我们希望它们能够协调 行为不同的参与方，甚至依据动态确定的信息来协调，该怎么办?

陷阱 委托和反应式多方计算是重要的协调机制，但并不完全够用。一些框架提供了底层机制，供程序员指定特定参与方的操作，但这些机制很容易被误用。举例来说，下面是一个 Obliv-C 程序，其中 <代码> 仅由参与方 *X*执行：

```
如果 (ocCurrentParty()  ==  X) {  <代码>  }
```

其他 Obliv-C 结构也会接受参与方标识符，以便将执行限定在特定参与方上，例如，`readInt`会从所标识参与方的本地存储中读取数据。使用此类结构时必须谨慎。假设 A 想要共享在其输入`a`上计算`f` 所得的加密结果。下面这段用于实现该目的的代码存在一个错误：

```
1  int a;2    readInt(&a, "input.txt", A);3
ob
fe
 /
```

由于第 4 行的条件表达式，第 5 行的共享操作会在 A 上触发发送，但 B 上没有对应的接收，导致 A 永远阻塞，等待 B 的响应。我们通过删除第 4 行的条件表达式来修复此问题，这样双方都会调用`feedOblivInt`。

```
1  int a;2    读取整数(&a, ..., A);3
ob
```

代码不再在 A 上挂起，但现在出现了另一个问题：未定义行为。第 2 行的`readInt`调用会在 A 上初始化`a`，但不会在 B 上初始化。`f(a)` 调用会导致 A 和 B 都读取`a`中包含的值，从而在 B 上引发未定义行为。解决方案是在 B 上执行时为`a` 提供一个虚拟值。[1]

```
1  int a;2    读取整数(&a, ..., A);3    int fa =
oc
fe
```

仍然存在风险：`f` 不得在参与方之间进行任何通信，否则我们将遇到另一个协调错误，就像第一个示例中的错误一样。

三**方（或更多方）协调** 为$N \geq 2$ 个参与方编写多方计算时，协调会变得更加棘手。通常，参与方可能有数十个甚至更多，且重叠的参与方集合之间会发生交互。每个参与方的角色都可能随时间变化，而且可能取决于先前的计算。这些复杂性或许正是 PICCO [40], 、Frigate [26], 和 SCALE-MAMBA [2]等N方语言甚至不提供底层协调机制的原因。

Wysteria [28, 30] 的独特之处在于，它为安全的N方协调提供了更高层次的支持。以下是第 2.2 节中用 Wysteria 编写的整数共享示例。

```
1 let 绑定 a =并行({A})= 读取() in2    let 绑定 fa =并行
({A
4
et
```

虽然所有参与方都在本地执行同一个程序，但 `let x =par(P)= e in` .....这种标注表示只有 `P` 中的参与方应执行`e`；其余参与方将跳过该表达式。参与方标注的表达式可以防止协调错误。例如，第 1 行由 A 创建的变量`a`，只有在 A 单独处于作用域内时才能被访问，例如第 2 行所示。Wysteria 的

[1]我们也可以改为确保在 B 上将 a 初始化为一个虚拟值。这种方法可行，但在处理复合类型（例如整数数组）时，需要在 B 上为 A 的所有输入分配内存，并用虚拟值初始化这部分内存。

类型系统会拒绝尝试以其他方式操作的程序。当变量对多个参与方可见时，Wysteria 的设计确保所有参与方都同意其内容。

Wysteria 还提供了一种用于指定多方计算（MPC）的注释：`let x = sec(P) e in ...`...表示参与方 `P` 应作为多方计算共同执行 `e` 。运行时执行到这里时，Wysteria 解释器会将`e` 转换为电路，并使用 GMW 协议执行该电路，将结果揭示给所有参与方。输入被指定为线束，通过`wire P e`语法创建；它们本质上是从参与方到值的映射，其中映射到的值仅对作为键的参与方可见。参与方将其输入打包到一个捆绑包中（如第 4 行所示），然后可在`sec` 表达式中通过数组索引表示法访问这些输入（第 5 行获取了参与方 A 的输入）。程序员可以通过`makesh` （第 5 行）将值标记为份额，而不是立即揭示最终结果；之后可在后续的`sec` 表达式中再次引用这些份额（由同一组参与方执行），继续对其进行多方计算（稍后将对此进行说明）。同样，参与方注释可防止访问不存在的值。此外，使用`sec` 可确保所有参与方在多方计算处进行同步，从而避免死锁或挂起。

虽然 Wysteria 的协调支持优于其他框架，但仍有一些局限。首先，尽管 Wysteria 的参与方集合是值，但对它们的操作十分有限，因此给定完整程序后，总能静态地确定这些集合。（更多信息请参见附录 A.3。）协调动态选出的参与方对安全性（例如，随机选择参与方以防止作弊）和交互（例如，纸牌游戏中的玩家集合）都很重要。此外，Wysteria 不区分计算方和输入输出方，因此无法支持委托，即使参与方集合是静态的也是如此。Wysteria 的份额无法在不解密的情况下重新共享给其他参与方。

3 Symphony：表达能力强、协调式多方计算

Symphony 是一种领域特定编程语言，通过推广标准的 SIMD 风格视图，用于表达由$N \geq 2$ 个参与方执行的多方计算。它借鉴了 Wysteria 解决协调问题的方法，提供了类似的安全性保证，同时显著提升了语言的表达能力和效率。本节通过示例介绍 Symphony；接下来的几节将对其进行形式化阐述，并证明其安全性优势。

3.1 基础

Symphony 是一种动态类型的函数式编程语言，支持整数、对、变体（和）类型、列表、let 绑定、模式匹配、（递归）高阶函数，以及（可变）引用和数组。

回顾一下第 2.1 节中用 Obliv-C 实现的百万富翁示例。Symphony 中的类似实现如下：

```
1 主体 AB 2 定义 main () = 并行 {A,B} 3 let 绑定 a = 并行
{A
in
{A
中
A,
```

按照惯例，所有参与方运行同一个程序（从`main`开始）。

并行代码块 Symphony 使用 并行 代码块，允许部分而非所有参与方执行程序的一部分。例如，在第 3 行遇到并行 `{A}`... 时，集合中列出的参与方 A 会执行后续代码...。在这种情况下，参与方 A 会从其本地文件系统中的文件输入。`txt` 读取一个整数。不属于并行集合的参与方会跳过代码块，并返回一个不透明值 ★；如果对该值进行计算，就会导致错误（因为该参与方没有可用的实际值）。

数据位置 由于 Symphony 程序本质上是分布式的，数据位于特定参与方的主机上。一个重要的不变量是，当不同主机访问同一个程序变量时，它们看到的逻辑值必须相同；否则，整个程序都会失败，因为至少有一台主机会看到 ★。（Symphony 还提供了类似 Wysteria 的捆绑 抽象，允许不同参与方使用同一个变量来保存不同的值。）因此，参与方能够自然地协调，从而避免错误结果、死锁以及第 2.2 节提到的程序挂起等问题。

份额 在 Symphony 中，加密值称为份额，我们可以将其视为在特定参与方集合之间进行的秘密份额共享。（Symphony 实现了 GMW 和 Yao 后端。）我们使用 share$[\phi, \tau : \mathrm{P} \rightarrow \mathrm{Q}]$ v，将当前位于 P 的值 v 在参与方 Q 之间进行秘密共享，其中 ϕ 是多方计算协议（例如 gmw），τ 是份额内容的类型。通常，P 是单个参与方，而 Q 是一个集合。例如，第 5 行将 A 的值 a 秘密共享给 A 和 B，并将份额存储在 v1 中。这维持了位置不变量：参与方 A 和 B 对 v1 的逻辑值达成一致——它是这两个参与方之间对同一值的份额。share 操作要求 $\mathrm{P} \cup \mathrm{Q}$ 中的所有参与方当前都在执行（由第 2 行的 par 确保），这样 P 才能向 Q 中的每个参与方传送其份额，并确认它们已准备好接收。每个本地执行的参与方都会跟踪作用域内并行块集合的交集，以执行此检查。只要共享参与方共同执行操作，份额就可以像普通值一样进行运算。例如，第 7 行比较共享的整数，并将结果存储在 ge 中（它也是一个份额）；A 和 B 都会对其逻辑值达成一致。

揭示 当 共享 将其参数从P 处转换为Q处的份额时，揭示 则将P 处的份额转换为Q处的明文。第 8 行将 A 和 B 处的共享ge 转换为明文值，并发送给这些相同的主机。与共享一样，$P \cup Q$ 中的所有参与方都必须执行该代码块，否则会引发错误。

推理 尽管程序实际上是在多台相互通信的机器上执行的，Symphony 允许程序员将代码视为在单台机器上逐行执行，以此进行推理；我们将这种方式称为“单线程解释”。我们通过第 6 节中的三个定理证明了这一点，这些定理的证明依赖于定位数据和份额上的关键不变量。第一个定理指出，当且仅当程序在实际执行中终止时，它才会在单线程解释中终止（这意味着不会发生死锁）。这有助于调试；例如，如果程序在单线程模拟中使用合成输入能够终止（我们的实现提供了这种模拟），且终止与输入数据无关，那么就可以保证该分布式程序在使用秘密输入进行实际执行时也会终止，且绝不会发生死锁。第二个和第三个定理针对因 Symphony 检测到运行时错误而中止的执行，确立了类似的对应关系——例如，某个参与方访问了包含 的变量，或执行了参与方缺失的共享 操作——它们实质上表明，当且仅当实际执行中的至少一名参与方会检测到同一错误时，单线程执行才会检测到此类错误。这使程序员能够借助单线程解释中简单得多的失败语义，推理实际分布式执行中的失败。（可以开发静态类型系统来彻底避免失败，但要使其表达能力足够强将极具挑战，因此这仍是未来的工作。）

3.2 高级协调

Symphony 通过一等份额和一等参与方集合，结合标准函数式编程构造，支持高级协调模式。

一等份额虽然以往的语言支持将份额作为数据，但会限制份额的创建及其操作，也就是说，份额并非真正的一等对象。Symphony 更为灵活，允许委托和重新分享份额。为支持委托，我们可以编写共享$[\phi, \tau : \{p\} \to Q]\, v$，让每个 IO 参与方$p$ 将其输入分享给计算方Q，由它们计算结果并将最终结果分享给每个p。与以往的系统不同，p 无需属于Q。为支持重新分享，我们将份额v 作为输入传给共享$[\phi, \tau : P \to Q]\, v$，而不是传入明文值。

一等参与方集合 通过使用一等参与方集合，委托和重新分享变得更加强大——像`{A、B}` 这样的集合是运行时值，而不是静态注解。下面的 Symphony 程序委托参与方 E 和 F 来计算参与方 A 是否是参与方 A–D 中最富者。

Symphony：借助协调实现的表达性安全多方计算

1 主体 AB C D E F 2 3 -- read input at p, delegate to all in Q 4 定义

rea

--

8

映射

sha

=

gm

delegateShares 函数接收两个参与方集合作为输入：P 是输入输出方集合，Q 是计算方集合。该函数返回一个份额列表，其中每个份额对应P中的一个参与方，并位于Q中的参与方处。它通过对参与方P调用map 来计算这一结果，并使用readShare Q读取输入，将其共享给Q。

第 12 行为参与方{A,B,C,D} 计算份额，并将其委托给{E,F}。随后，第 14 行指示{E,F} 对这些份额进行折叠，检查 A 的输入是否始终为最大值。第 15 行将结果揭示回输入输出方。

此处未展示，Symphony 允许通过case对参与方集合进行解构，这对于在参与方之间进行动态协调很有用，例如选举委员会来执行委托式多方计算。数据位置不变式允许在运行时可靠地协调参与方的选择。

LWZ 重新分享、委托以及一等公民的参与方集合，对于实现 Laur、Willemson 和 Zhang [23]提出的高效安全洗牌 计算至关重要。以下是 Symphony 中该计算实现的一段代码（完整代码见清单 4，附录 A.2）。

1 def 使用.....打乱 Q S Q 的份额 = 并行 (Q \/ S) 2 let S 的份额 =share [g

m

(sl

Q 和S 均为参与方集合，Q 的份额 是一个位于Q的int 份额数组。第 2 行将Q 的份额 重新分享给参与方S，第 3 行调用shuffle 在S处打乱这些份额，然后将结果重新分享回Q。对重新分享的语言支持是 Symphony 独有的，也是实现 LWZ 的关键。据我们所知，由于协调方面的挑战，其他多方计算语言都无法表达这一协议。

3.3 与相关工作的比较

Symphony 在表 1 中与相关的多方计算框架进行了（部分）比较。在表达能力方面，它胜过了所有这些框架。与许多框架一样，它支持超过 2 个参与方、反应式多方计算和同步随机性。与 Wysteria 一样，它通过 par 块支持安全协调，但对其进行了推广：程序可以对模式匹配

参与方集合（不只是构造它们），并且可以直接对加密值进行计算（无需 `sec` 块），还可以委托和重新分享。我们证明的 Symphony 分布式语义与单线程语义之间的对应定理（第 6 节），与 Wysteria 的元理论结果相似；不过，定位数据上的不变量是 Symphony 独有的，且这些定理也针对 Symphony 的动态类型系统进行了调整。在实现方面（第 7 节），Symphony 同时支持混淆电路（适用于 2 个参与方）和秘密份额（适用于 3 个或更多参与方）。这些特性结合起来带来了良好的性能。

表 1 中列出的一项相关工作与其他工作截然不同，它是 Viaduct [1]，该系统将类Java语言编译为安全分布式程序。Viaduct 的编程模型比 Symphony 更高层；它根据指定为信息流标签的安全策略 [31]，合成计算/通信模式，而不是直接指定这些模式。它还支持 MPC 以外的密码方案（例如，承诺方案和零知识证明）。由于 Symphony 将主体集合、捆绑包和平行块作为一等概念，因此它支持比 Viaduct 更丰富的协调模式（例如 LWZ）。Viaduct 没有与 Symphony 的可靠性保证（第 6 节）相对应的结果，但提供了专门用于指定安全策略的机制，这些策略可能涉及降密和背书。从本质上说，Symphony 采用“理想世界”方法，依赖程序员确保程序不会泄露过多信息。

Symphony 对“广义SIMD”的语义与编舞语言 [8, 9, 10, 25] 的语义有相似之处。编舞程序在概念上是顺序执行的，它们将发送和接收操作合并为单个表达式，从而确保这些操作始终匹配。Pirouette [18] 是一种类型化编舞函数式编程语言，它证明了类型良好的程序在分布式部署时按设计即可无死锁。得益于静态类型和静态参与方注解，Pirouette 能够证明编舞与其分布式部署之间的强元理论性质。

4 λ-Symphony：语法与语义

λ-Symphony 是一种精简的核心语言，涵盖了 Symphony 的基本特性。本节介绍其语法和单线程(ST)语义。

4.1 语法

λ-Symphony 的语法如图 1 所示。为简化形式语义，该语法采用一种管理范式 (ANF)，这意味着大多数表达式形式都直接作用于变量x，而非子表达式e，实际实现中也是如此。我们将原子表达式a 单独列为完整表达式e的一个子类别；前者只需一个“小”步即可求值为最终结果。

大多数语法形式都是标准的。二元运算作用于整数或份额（例如 $+$、 $\times$），也可以作用于参与方集合（例如 $\cup$）。条件表达式 $x\,?\,x \diamond x$ 对应于多路复用器表达式（在 Symphony 中写作 `mux if` ）。对可以通过投影访问（例如， π_1 $\langle 1,2 \rangle$ 求值结果为 1），而和类型（又称变体或标记联合）

$i \in \mathbb{Z}$	整数
$A, B, C \in$ 参与方	参与方
$m, p, q \in$ pset $\triangleq \wp$(参与方)	参与方集合
$x, y, z \in$ var	变量
$\odot \in$ bop	二元运算 $(+, \times, \cup, \ldots)$
$a \in$ 原子 $::= x$	变量引用
$\mid i$	整数字面量
$\mid p$	参与方集合字面量
$\mid x \odot x$	二元运算
$\mid x ? x \diamond x$	条件表达式
$\mid \iota_i x$	和类型注入
$\mid \langle x, x \rangle$	二元组创建
$\mid \pi_i x$	二元组投影
$\mid \lambda_z x.\, e$	（递归）函数定义
$\mid$ 引用x	引用创建
$\mid\ !x$	解引用
$\mid x := x$	引用赋值
$\mid$ 读取	读取整数输入
$\mid$ 写入x	写入输出
$\mid$ 共享$[x \to x]\ x$	共享加密值。
$\mid$ 揭示$[x \to x]\ x$	揭示加密值。
$e \in$ 表达式 $::= a$	原子表达式
$\mid$ case$x\ \{\lceil x.e\}\{\lceil x.e\}$	和类型、幂集的消去
$\mid\ x\ x$	函数调用
$\mid$ 并行$x\ e$	并行执行
$\mid$ let 绑定 $x\ = e$ in e	*let* 绑定

图 1 λ-Symphony 形式语法。

可以通过模式匹配访问（例如，case $\iota_1\ 0\ \{y.e_1\}\ \{y.e_2\}$ 求值结果为 e_1，其中 y 被替换为 0）。参与方集合也通过 case 访问，并像列表一样处理——第一个分支处理 $\varnothing$ case，第二个分支则绑定两个变量，一个代表选中的参与方，另一个代表集合中的其余部分。递归函数写作 $\lambda_z x.\, e$；函数体 e 可以通过变量 z 引用自身。 λ-Symphony 还支持可变引用以及输入/输出原语。 λ-Symphony 不对列表或捆绑包建模，因为它们很容易编码；我们将在附录中说明具体方法。与 MPC 相关的构造 par、share 和 reveal 与 Symphony 中的对应构造一致；后两者省略了输出类型和协议注释（这些对实现很有用，但对形式化建模并非必要）。正如第 7 节所述，Symphony 的实现还推广了 λ-Symphony 的其他方面；例如，它允许共享任意类型的值，并对加密和类型进行分类讨论。

4.2 概述

λ-Symphony 的 ST 语义将所有参与方建模为仿佛在锁步执行。我们证明，ST 语义忠实地刻画了第 5 节所述的分布式（DS）语义；根据该语义，参与方可以独立行动——

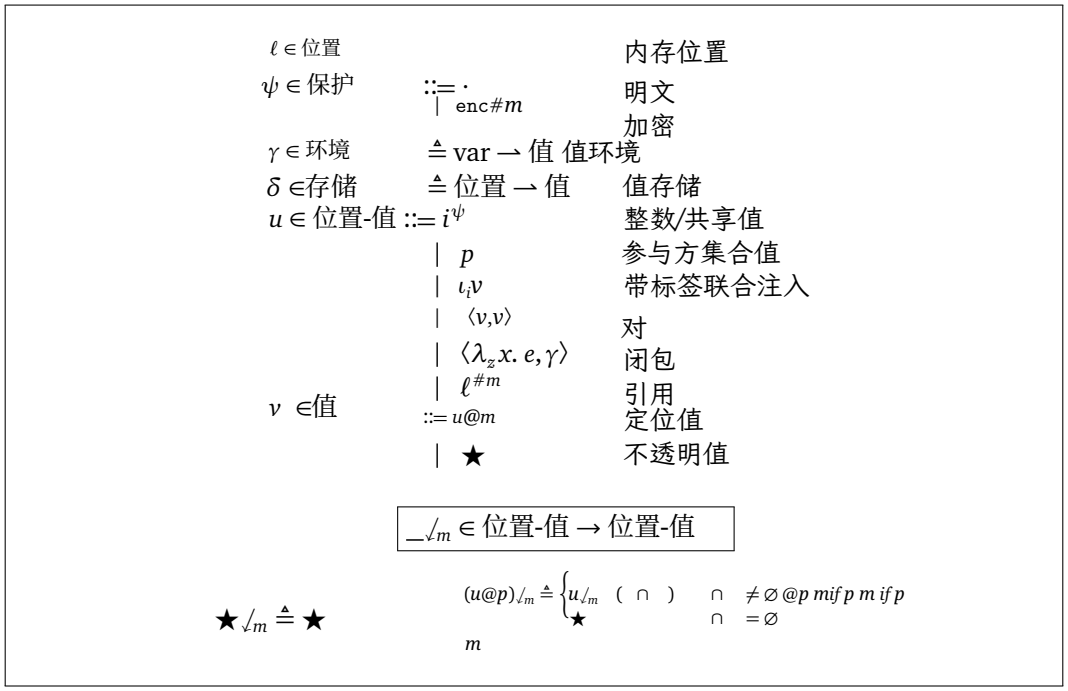


图 2 λ-Symphony 形式语义中使用的定义和元函数（部分）。

立。因此，ST 语义可作为 λ-Symphony 形式推理的基础，例如用于推理正确性和安全性。

主要判断 $\varsigma \longrightarrow \varsigma$ 是配置 ς之间的归约关系。一个配置是一个五元组，由当前模式m、环境 γ、存储 δ、栈 κ和表达式e组成。模式是并行计算e 的参与方集合；我们称参与方$A \in m$ 在一次计算中在场。根据图 2，环境是从变量到值的偏映射，存储是从内存位置到值的偏映射；稍后我们会讨论栈。主要判断使用 $\gamma \vdash_m \delta, a \hookrightarrow \delta, v$，它定义了原子表达式$a$ 归约为值v 的过程。这两个判断的部分规则见图 3；完整规则集见附录。

4.3 值

值v 有两种形式之一：$u@m$ 表示定位值u 仅$A \in m$可访问，例如，因为它是以模式m求值e 的结果；而 ★ 则是不透明值，既不可知也不可访问。图 2 定义了定位值，包括数字i^{ψ}、参与方集合p、和类型 $\iota_i\, v$、对 $\langle v,v\rangle$ 、递归函数 $\langle \lambda_z x.\, e, \gamma\rangle$ （其中包含闭包环境 γ），以及内存位置（即指针） $\ell^{\#p}$。除注释 ψ 和$\#p$外，这些都是标准形式。

注释$\#p$ 用于指明p 这些参与方是共同创建者，即它们共同创建了所引用的内存；而 ψ 则指明协议，该协议作用于带注释的整数：$\cdot$ 表示明文值（我们只写i，此时注释 ψ 为$\cdot$）；而$\mathtt{enc}\#p$ 表示由参与方$B \in p$ 共同持有的加密值（即“共享”）。因此，值 $1^{\mathtt{enc}\#q}@q$

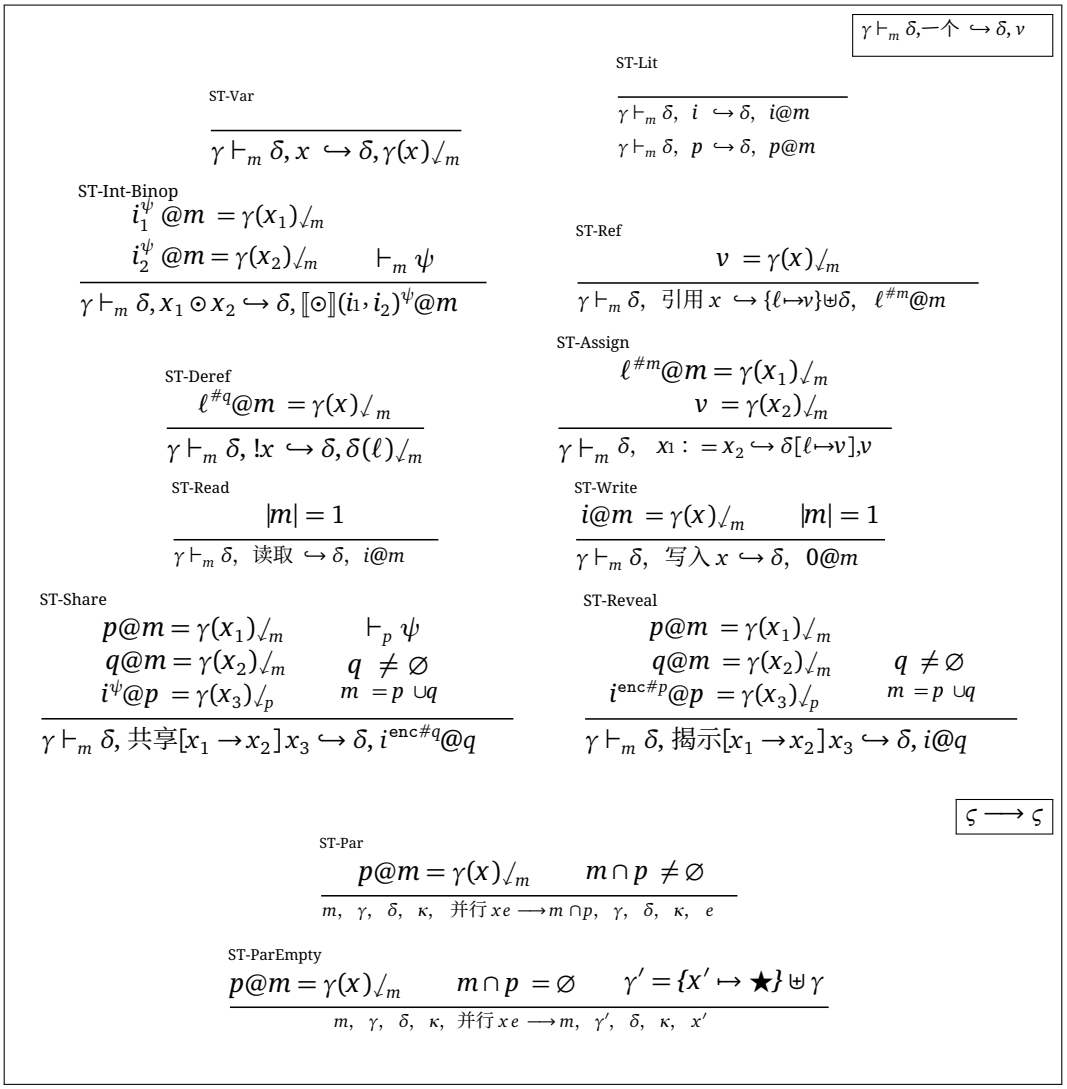


图 3 λ-Symphony 单线程语义，选定的规则。

可读作“一个整数1，在参与方q之间加密（即秘密共享），并可由参与方q访问。”第一个q 表示该值在哪些参与方之间共享（在创建共享时确定），第二个q 表示哪些参与方可以访问该值（由外围的并行块确定）。位置注释仅在 ST 语义中使用，以模拟多个参与方的存在；在分布式语义和最终执行中则不会用到。另一方面，enc#q 和#p 注释用于在分布式执行期间检测未能正确协调的有缺陷的程序，例如，若A单独尝试对由A 和B共同持有的共享值进行算术运算，或若仅A尝试写入其与B共同创建的引用。

4.4 操作规则

现在我们来考虑一些操作规则。

字面量、变量、绑定、计算 规则 ST-Var 从环境中获取变量的值，并通过 $\gamma(x)\downarrow_m$ 将其重新定位*(re)locatesit* 到当前模式 m。函数$\downarrow_m$在图 2 中给出。对于值$u@p$，$\downarrow_m$ 会将其重新定位到$p \cap m$；除非交集为空，在这种情况下该值不可访问，因此变为 ★。重新定位是一种深层操作；$u@p\downarrow_m$ 还会将内容u 重新定位到 $u\downarrow_m$，这一过程会递归遍历u的子项。重新定位确保获取的值与兼容 m。当值v 可供某个参与方集合$p \subseteq m$访问时，该值v 就与参与方集合m 兼容。与当前模式的兼容性是所有规则的通用不变量。规则ST-Lit 为整数和主体型字面量确定类型，并为其标注（兼容的）位置m。重新定位的完整定义见附录。

使用栈 κ 管理通过 let 绑定定义的变量。栈可以是空栈 $\top$，也可以是由栈帧组成的列表：$\langle \text{let x} = \square \text{ in e} \mid \text{m}, \gamma \rangle :: \kappa$。为了求值 $\text{let x} = \text{e}_1 \text{ in e}_2$，我们会压入栈帧 $\langle \text{let x} = \square \text{ in e}_2 \mid \text{m}, \gamma \rangle$，并将活动表达式设为 e_1（规则 ST- Let-Push，未显示）。当表达式求值为值 v 时，最顶层的栈帧 $\langle \text{let x} = \square \text{ in e}_2 \mid \text{m}, \gamma \rangle$ 会被弹出，随后使用保存的模式 m 和环境 γ 继续求值 e_2，并将 x 映射到 v（规则 ST-Let-Pop，未显示）。

规则ST-Int-Binop 处理整数上的算术运算。该规则展示了所有消去规则共有的另一项不变量：在模式m 下运行时，要对某个值进行计算，该值必须能被m中的所有参与方访问。我们可以从这样的前提中看到这一点：$i_1^{\psi} @m = \gamma(x_1)\downarrow_m$。这些前提将运算所针对的变量定位到当前模式$m$，并确保该值的位置也为$m$，即所有参与方都能访问待计算的值。这样可以确保这些参与方在分布式环境中各自使用自己的存储、环境等运行时，对结果达成一致。对于这条规则，我们还会确保两个整数使用相同的协议 ψ，且该协议与模式m兼容，记作 $\vdash_m \psi$。当 ψ 为明文时，兼容性成立；当其为 enc#m 时也成立，即i 是当前所有参与方共同持有的共享值。分布式语义也会使用兼容性检查来确保各参与方保持同步。

并行模式 在操作语义上，$\text{par}\, x\, e$ 会在模式 $m \cap p$ 下计算e，其中$p@m = \gamma(x)\downarrow_m$；也就是说，只有$p$ 中同时也 存在于m 中的参与方才会运行e。当$m \cap p$ 非空时，规则 ST-Par 会指示e 在细化后的模式下计算。如果$m \cap p$ 为空，则根据规则ST-ParEmpty，跳过e 并返回 ★。[2] 请注意，由于栈会跟踪每个栈帧的模式，当当前表达式计算完成并弹出栈帧时，旧模式将恢复。

以下是并行模式与变量访问如何相互作用的示例。

并行 {A,B} let 绑定 x = 并行 {A} 1 i
n let 绑定 y =并行 {B} x in let 绑定
z =并行 {C} 2 in x

外层并行 {A, B} 按照规则 ST-Par，在模式 {A, B} 下计算其主体。接下来，根据规则 ST-LetPush、ST-Par 和 ST-Int，我们在模式 m = {A, B}∩{A} = {A} 下计算 1；按照规则 ST-LetPop，将结果 1@{A} 绑定到 γ 中的 x。接下来，根据规则 ST-

[2]由于 ★ 不是表达式——它是一个值——我们会返回一个新变量，以及一个将该变量映射到 ★ 的环境。

LetPush、ST-Par 和 ST-Var，我们在模式 m ={B} 下计算 x。根据规则 ST-Var，我们取回 x 对应的值 1@{A}，然后 $\downarrow_{\{B\}}(1@\{A\})$ 得到 ★ 作为结果，并按照规则 ST-LetPop 将其绑定到 γ 中的 y。这个结果是合理的：参与方 B 读取变量 x，而其内容只有 A 才能访问，因此它所能做的就是返回不透明值。最后，根据规则 T-ParEmpty，par {C} 2 的计算结果为 ★，因为 m ={A, B} ∩{C} =∅。按照规则 ST-LetPop，这个 ★ 结果被绑定到 z；最终结果 x 在模式 m ={A, B} 下求值为 $\downarrow_{\{A,B\}}(1@\{A\}) = 1@\{A\}$，依据规则 ST-Var。

引用 规则 ST-Ref会创建一个新的引用，并返回一个带位置指针，标注创建它的参与方。规则ST-Deref 会获取位于当前模式m中的引用，并返回与m兼容的指向内容。规则ST-Assign会用新值更新存储并返回该值，但仅适用于 $\ell^{\#p}$引用，其中$p = m$，即当前模式。为什么？请看下面的例子。

并行 {A,B} let x = ref 0 inlet = (并行{A} x := 1) inlety = !x in...

变量x 最初包含一个引用 $\ell^{\#\{A,B\}}$，因为它是在模式$m = \{A,B\}$的上下文中创建的。随后，在下一行的并行表达式中，A 对x 进行了赋值。根据规则ST-Assign，引用的创建者$\#\{A,B\}$ 必须与模式m 相符才能继续，但由于m 是$\{A\}$，程序因此卡住。这是合理的，因为若继续执行，A和B对计算的视图就会失去同步。作为分布式语义的一部分，我们分别在A和B 上运行该程序时，在A上会执行赋值，而在B 上则会跳过赋值。因此，在A 上，y的值会是1，而在B上则会是0。如果程序在...处的后续部分根据y 进行分支，并在其中一个分支中执行一些多方计算构造、而在另一个分支中不执行，那么两个参与方就会更加失去同步。

输入/输出 规则 ST-Read与 ST-Write 处理输入/输出。它们要求模式为单例参与方，并将得到的值定位于该单例参与方。这对于确保所有参与方就共享变量的内容达成一致非常重要。在语义中，readnondeterministically 返回任意整数，这对从主机环境中读取特定整数作为输入的行为进行了过度近似。

多方计算中，参与方 A 使用 share[$x_1 \to x_2$] x_3 语法，在参与方 q 之间创建加密值（即份额），并由规则 ST-Share 处理。变量 x_1 是输入参与方 p 的集合；变量 x_2 是将持有份额 q ≠ ∅的（非空）参与方集合；而 x_3 是要共享的值，由输入参与方 p 知晓；如果 x_3 已经是加密份额（在 p 之间共享），则此操作会在 q 之间重新分享该份额。输入参与方 p 和份额参与方 q 必须全部包含在模式 m 中，且不能包含其他参与方（因此 m =p ∪q）。重要的是，p ⊆q 并非必要条件——这意味着参与方 p 可以将安全计算委托给参与方 q，而无需亲自参与其中。重新分享和委托都是 Symphony 的关键优势，现有多方计算语言并不具备这些特性。LWZ 中的 share 表达式（清单 4，附录 A.2）充分利用了这两项特性。

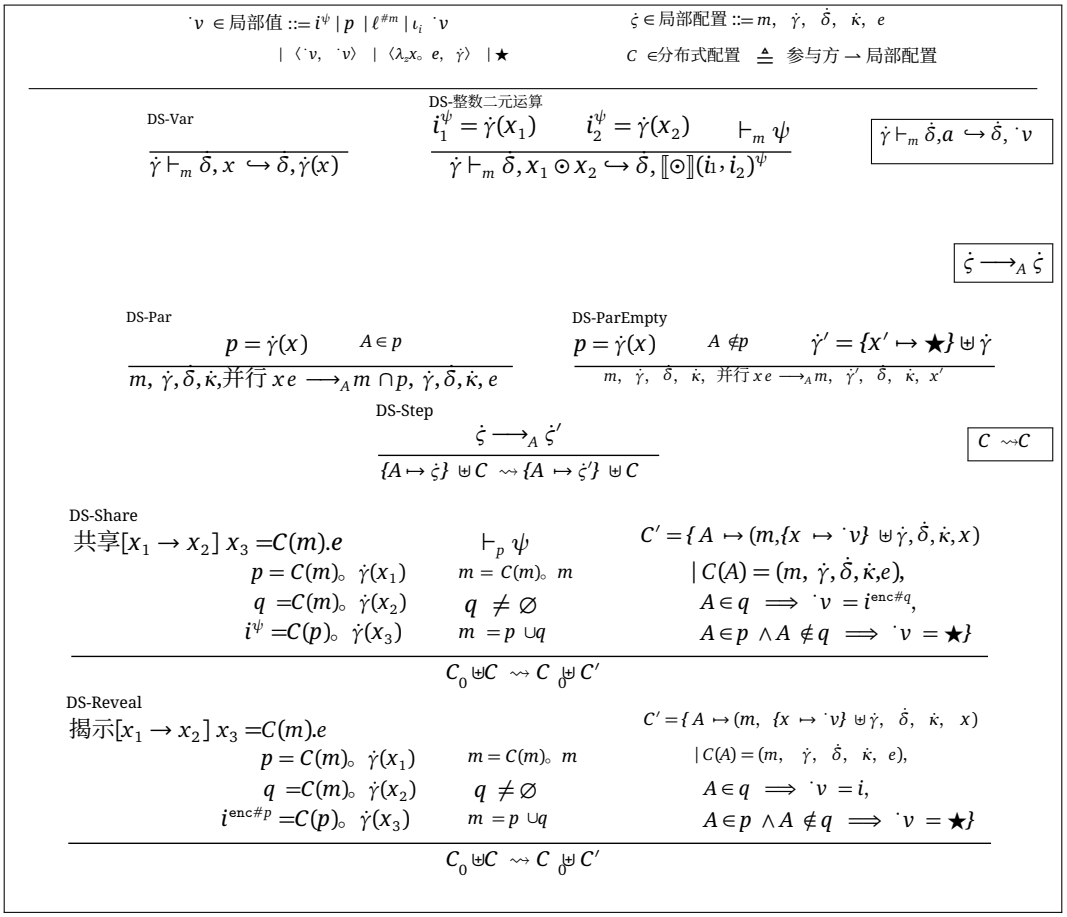


图 4 λ-Symphony 分布式语义，选定的规则。请注意，判断 $\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}, \dot{v}$ 由 $\dot{\varsigma} \longrightarrow_A \dot{\varsigma}$ 判断中的省略规则 DS-LetPop 引用，与图3中的单线程语义类似。

根据规则ST-Reveal，$\text{reveal}[x_1 \to x_2]\, x_3$ 会将参与方p 之间共享的加密值揭示为参与方$q \neq \varnothing$处的明文结果，其中x_1 的计算结果为p，x_2 的计算结果为q，而x_3 的计算结果为加密值。所有参与方p （x_3 在它们之间共享）都必须在场，值的接收方q也必须在场，且不能有其他参与方。

5 分布式语义

本节介绍 λ-Symphony 的分布式（DS）语义，对多方计算所需的通信与协调进行建模。下一节将证明 ST 语义与 DS 语义之间的对应关系。

5.1 配置

一个分布式配置C 汇集了多方计算中各个参与方的执行状态。如图 4 顶部所示，它由一个从参与方到局部配置 ς的有限映射构成；局部配置是由以下五项组成的五元组：(1) 模式m，(2) 一个局部

环境 $\dot{\gamma}$，(3) 一个局部存储 δ，(4) 一个局部栈 $\dot{\kappa}$，以及 (5) 一个表达式e。局部环境、存储和栈与其 ST 对应规则相同，只是其中包含的不是值v，而是局部值 $\dot{v}$，后者不带位置注释$@m$。

对于希望联合执行程序 e 的参与方集合 m，初始配置C_0 会将每个参与方 $A \in m$ 映射到一个局部配置 $(m, \varnothing, \varnothing, \top, e)$，其中 $\varnothing$ 为空函数（用于空环境和空存储），$\top$ 为空栈，而e 是源程序。请注意，每个参与方都会在其局部配置中跟踪全局模式 m。

5.2 操作语义

DS 语义$C \rightsquigarrow C'$ 使用辅助判断 $\dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}, \dot{v}$ 和 $\dot{\varsigma} \longrightarrow_A \dot{\varsigma}$；这些判断的部分定义见图 4。主规则DS-Step用于独立执行单个参与方，不受其他参与方影响。该规则选取某个参与方A的局部配置ς，将其推进到 ς'，然后将其纳入分布式配置中。只要A 的活动表达式e 不是 share 或 reveal，就可以使用此规则，因为这两种操作需要多个参与方之间同步。这两种情况分别使用规则DS-Share 和DS-Reveal，下文将对此进行讨论。

非同步表达式 关系 $\dot{\gamma} \vdash_m \delta$（一种 $\hookrightarrow \delta, \dot{v}$）的规则与 ST 语义的规则基本相同，但它们作用于无位置数据。图中展示了两个例子。ST-Var 规则的结论通过 $\gamma(\mathrm{x})\downarrow_m$ 将结果定位到 m，而 DS-Var 规则的结论则只返回 $\dot{\gamma}(\mathrm{x})$。类似地，ST-IntBinop 规则的前提要求 $\mathrm{i}_1^{\psi}\ @\mathrm{m} = \gamma(\mathrm{x}_1)\downarrow_m$，而 DS-IntBinop 规则的前提只要求 $\mathrm{i}_1^{\psi} = \dot{\gamma}(\mathrm{x}_1)$。对于消去形式，ST 规则中的位置不匹配，在 DS 规则中会导致尝试消去 $\bigstar$ 失败。例如，若 ST-IntBinop 规则因 i_1^{ψ} 的位置不是 m 而是 $\mathrm{p} \subset \mathrm{m}$ 而失败，那么对于参与方 $\mathrm{A} \in (\mathrm{m} - \mathrm{p})$，DS-IntBinop 规则也会失败，因为这些 $\dot{\gamma}(\mathrm{x}_1) = \bigstar$，无法与另一份额相加。两条规则中都包含检查 $\vdash_m \psi$，以防止尝试相加不兼容的份额。同样，DS-Deref 和 DS-Assign 规则（未显示）保留了 ST 版本中的检查，即引用所有者必须与 m 兼容。

判断 $\dot{\varsigma} \longrightarrow_A \dot{\varsigma}$ 对应于 ST 判断 $\varsigma \longrightarrow \varsigma$，其中注释 A表示执行方本地参与方。除了处理并行$[x]\ e$的规则外，这两个判断的规则本质上相同。只要 $A \in p$，规则 DS-Par 就会求值表达式 e，其中 $p = \dot{\gamma}(x)$，并将全局模式更新为 $m \cap p$，与 ST 语义的做法相同。规则 DS-ParEmpty处理 $A \notin p$的情况，因此会跳过 e，并保持全局模式 m 不变，求值得到结果 x'；这是一个新变量，绑定到 $\bigstar$ 中的 $\dot{\gamma}'$。

同步表达式 规则 DS-Share 和 DS-Reveal 分别用于计算表达式的共享与揭示。这些表达式要求多个参与方之间进行同步，并将数据从一个参与方传输给另一个或多个参与方，因此这些规则会操作多个局部配置。但它们与对应的 ST 规则非常相似。

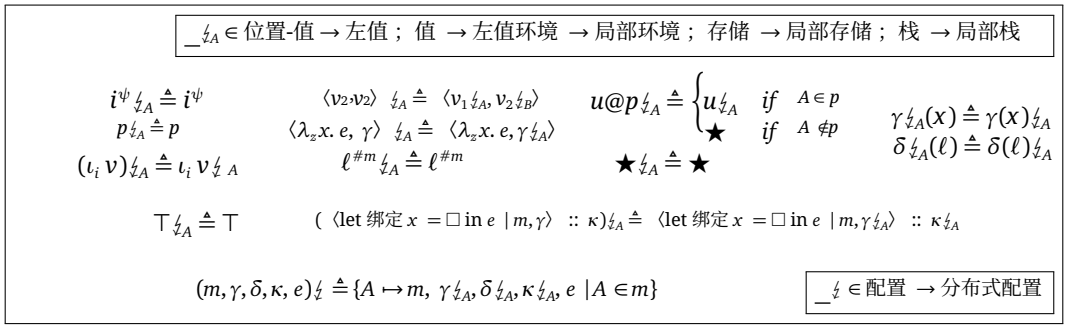


图 5 切片元函数；关联 ST 语义与 DS 语义。

在这些规则中，我们写作 C(m)，表示主体 A ∈ m 所映射到的配置集合。当我们写作 C(m).e = e′ 时，表示集合 C(m) 中每个配置的表达式分量（e）都等于表达式 e′。对于 DS-Share，e′ 是 share[$x_1 \to x_2$]x_3 ；对于 DS-Reveal，则是 reveal[$x_1 \to x_2$]x_3。同样，我们要求每个参与方的配置对 x_1 到 p 以及 x_2 到 q 的赋值保持一致，这两者共同构成约定的模式 m。对于 DS-Share，x_3 的赋值必须是一个整数，并且其协议 ψ 与 p 兼容：$\vdash_p \psi$；对于 DS-Reveal，所有共享参与方 p 对 x_3 的赋值必须是由这些参与方共同共享的加密整数。这些条件足以保证实际多方计算后端的共享和揭示操作能够成功完成。[3] 更新后的配置 C′ 与原始配置 C 相匹配，但会更新每个参与方 A∈m 的局部配置，使其表达式分量为 x；其中 x 是添加到存储 γ 中的新变量，用于映射到所传输的（明文或加密）整数；而对于满足 A∈p 且 A ∉ q 的共享参与方，则将其求值为 ★。

6 单线程可靠性

本节介绍围绕单线程可靠性的主要元理论结果。在这种意义下，我们可以根据 ST 语义来解释一个 λ-Symphony 程序，尽管它实际上会以分布式方式执行。证明见附录 B.2。

我们通过对单线程配置 ς 进行切片，将其关联到分布式配置，记作$\varsigma\downarrow$，其定义见图 5。模式m 下的每个参与方A ς 都映射到其本地 DS 配置，该配置由m、表达式e以及环境 γ、存储 δ和栈 κ 的切片版本组成，这些都属于 ς 且专属于A。切片体现了一个简单的想法：对于值$u@p$，如果$A \in p$ ，那么A 就可以访问u；但如果$A \notin p$ ，则无法访问；$\downarrow_A$ 的作用很像 $\downarrow_{\{A\}}$ ，但会剥除所有位置注释。

对于执行轨迹以范式（即终止状态）结束的程序，我们可以证明完整的对应关系。终止 ST 状态具有空栈，并且

[3] 请注意，在实际实现中，每个参与方 $A \in m$ 只需检查其自身对 m、p 和 q 的视图是否与 $m = p \cup q$一致；若不一致，正如下一节所示，它就会陷入停滞配置 ，并可报告多方计算因类型错误而失败。

已达到一个值；如果分布式语义状态中的所有局部配置均为终止状态，则该状态为终止状态（正式定义见附录）。

定理 6.1（ST/DS 终止对应性）。 若 $\varsigma \longrightarrow^* \varsigma'$，则对于任意 C，陈述（*1*）和（*2*）可相互推出：

1. ς' *is a terminal state and* $C = \varsigma' \lightning$　　*2.* $\varsigma \lightning \rightsquigarrow^* C$ *and* C *is a terminal state*

证明基于一个前向模拟引理，该引理确立了对于每个单线程执行，都存在一个与之兼容的分布式执行；还基于合流性，该性质确立了尽管分布式语义具有非确定性，其终止状态仍是唯一确定的。我们还证明了关于发散或卡住的执行的两个引理。

定理 6.2（ST/DS 强非对称非终止对应性）。 以下陈述成立：

1. 若 $\varsigma \lightning$（在 $\rightsquigarrow$ 下）到达卡住状态，则 ς（在 $\longrightarrow$ 下）到达卡住状态*2.* 若 ς（在 $\longrightarrow$ 下）发散，则 $\varsigma \lightning$（在 $\rightsquigarrow$ 下）发散

定理 6.2 并未排除 ς 卡住而 $\varsigma \lightning$ 永远不会卡住的可能性。考虑程序 let 绑定 x = 并行[A] <错误> in 并行[B] <无限循环>。在 ST 语义中，该程序会卡住。在 DS 语义中，A只会变为局部卡住，而B 会永远运行。4 我们证明，如果一个 ST 配置卡住，那么对于任意可达的 DS 配置，都存在一个可达的局部卡住状态：

定理 6.3（ST/DS 卡住状态可靠性）。 如果 $\varsigma \longrightarrow^* \varsigma'$ 和 ς' 卡住，那么对于每个 C，其中 $\varsigma \lightning \rightsquigarrow^* C$ 存在一个 C' 使得 $C \rightsquigarrow^* C'$ 并且 C' 处于局部卡住状态。

由此可知，如果对 ς 应用 ST 语义检测到运行时错误（即卡住），那么（假设调度器不存在病态行为）$\varsigma \lightning$ 的某个局部配置最终也会检测到运行时故障（即局部卡住），此时，实现便可以通知其他配置这一问题。

总之，对于终止和非终止程序，ST 语义与分布式语义相对应，但对分布式语义状态采用了局部的“卡住”概念。

7 实现

我们用 4K 行 Haskell 实现了一个 Symphony 解释器。该解释器可以在顺序模式下运行程序，用于原型设计和调试；也可以在分布式模式下运行程序，用于实际的多方计算。Symphony 在 λ-Symphony 的基础上增加了许多功能，包括布尔值和条件表达式；对对、和类型、列表、主体集合及捆绑包进行嵌套模式匹配；数组（支持 O(1) 查找的可变向量）；同步随机性；以及将常量隐式嵌入为份额。
Symphony 支持基于布尔电路的半诚实多方计算，包括使用 EMP 工具包 [36] 实现的 Yao 2 方协议，以及使用 MOTION [5]实现的 N 方 GMW 协议。

[4]这类示例也是我们无法证明完全双模拟的原因。

Symphony 解释器还通过允许 mux、case、share 和 reveal 在对、和类型及列表上递归运行，扩展了 λ-Symphony。它增加了 mux-case，用于对加密和类型进行分类讨论；加密和类型使用对表示： λ-Symphony 值 $\iota_0 v$ 表示为 〈true,〈 v,default〉〉 ，而值 $\iota_1 v$ 表示为〈false,〈default, v〉〉，其中每个组成部分都经过加密。值 default 用于允许 mux-case 对其两个分支都进行分类讨论，相当于一种多路复用器。default 的确切值在进行共享时根据 share[ϕ,τ: $P \to Q$] default 上的类型注解 τ 确定。

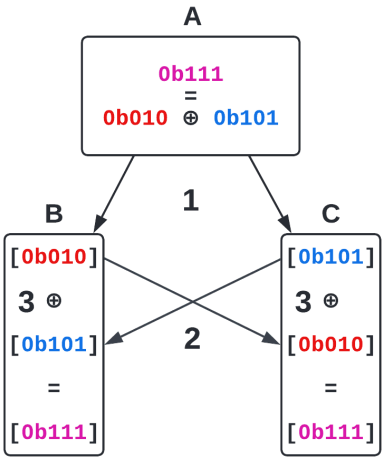
我们为 Symphony 实现了一个标准库（约 800 行代码），其中包含各种数据结构和协调模式，例如从主体集合初始化一个包，以及通过有界递归展开多方计算函数。附录 A.1 详细介绍了实现过程和标准库的内容。

Symphony运行时 EMP 和 MOTION 不支持委托或重新分享，而 MOTION 不支持反应式多方计算。Symphony 运行时由 2,000 行 Rust 代码实现，作为兼容层为多方计算后端补充必要功能。该运行时通过在 Symphony 参与方之间进行半诚实异或分享，支持委托和重新分享；并通过缓存加密值，支持反应式多方计算。

Symphony 中的加密值以抽象形式表示。使用各协议加密的值满足以下类 Java 接口的 Haskell 等效形式。

```
1 公共接口 Enc  <T> { 2 类型上下文; 3 4 // 将异或份额 `份额` 转换为抽象的加密
`T
8 v
Op
14
```

要向 Symphony 添加新协议，只需为该接口提供新的实现。第 2 行的 Context 类型声明并非合法的 Java 语法，其目的是让人联想到 Rust 中的关联类型或 Haskell 中的类型族。每种协议都会唯一确定一种类型，作为其 Context。该接口使解释器无需理会基于秘密共享的多方计算与基于电路混淆的多方计算之间的差异。例如，MOTION 后端通过逐步构建电路，并在请求解密时执行电路，来惰性实现多方计算。相比之下，EMP 则通过在门创建时进行混淆并执行，来急切实现多方计算。不过，这两种方法都可以实现上述接口。为此所需的增强，是本节余下部分的主题。



■ **图 6** 委托的运行时支持。

委托与重新分享 运行时通过 Symphony 参与方之间的半诚实异或共享，添加了对委托和重新分享的支持。例如，图 6 展示了参与方 *A*如何将其秘密比特串 0b111 委托给{*B*,*C*}。首先，*A* 为其输入生成两个随机异或份额，本例中为 0b010 和 0b101，并将它们分别发送给*B* 和*C*。其次，*B* 和*C* 将各自的份额转换为后端的原生加密表示。我们用方括号表示后端中加密的值，例如 [0b101]。第三，*B* 和*C* 使用多方计算后端计算其加密份额的异或，从而在{*B*,*C*}之间以原生加密形式得到*A*的原始输入。重新分享操作可以通过类似的方式实现，以类似的方式，将原生加密值转换为异或份额，然后遵循上文概述的委托过程。

委托和重新分享过程是通用的，将底层 MPC 后端视为黑盒，并且仅依赖标准功能。然而，通用地实现这些过程也无法利用特定协议之间转换时可用的优化。Symphony 并未实现所有最优的转换过程，但我们的方法并未妨碍我们这样做。MPC 文献中已对2方和*N*方协议的高效转换过程进行了详尽说明 [12, 5]。

反应式多方计算 Symphony 运行时还为 MOTION 后端添加了对反应式多方计算的支持。MOTION 中的 MPC 上下文称为 Party，是一个 C++ 对象，用于管理与 MPC 相关的全局状态。它包含有关当前执行的参与方、参与方总数、如何联系参与方（例如通过 TCP 套接字）、共享伪随机数生成器，以及由门和导线组成的二元电路的信息。创建或计算加密值时，Party 对象会创建一个新门并将其添加到电路中。实际的加密值是对新门输出导线的引用。程序员完成 MPC 后，会指示 Party 对象执行底层电路。此时，电路将使用 GMW 协议执行，并可从加密值中提取异或份额。

遗憾的是，Party 执行后，MOTION 不允许我们将 Party 对象重新用于额外的多方计算：该 Party 对象实际上已失效。继续执行多方计算的能力正是支持反应式多方计算所必需的；在反应式多方计算中，值会被解密，然后用于影响后续计算。

运行时通过缓存执行 Party 对象一次后产生的异或份额、销毁该对象，然后创建一个新对象，为 MOTION 增加了对反应式多方计算的支持。当我们对前一个 Party 生成的加密值进行计算时，会先将相应的缓存 XOR 份额作为输入提供给新的 Party，然后再继续计算。这个做法在概念上很简单，但要达到可接受的性能，需要进行大量工程改造。例如，我们必须修改

■ **表 2** 已实现的多方计算程序集合（详见附录 A.2）。#表示参与方数量。**特性** 表示实现程序所需的特性：P 表示一等参与方集合，R 表示反应式多方计算，$ 表示同步随机性，D 表示委托，S 表示重新分享。**Symphony**、**Obliv-C** 和**Wysteria** 表示所需的代码行数（包括 main 驱动代码），其中(+*K*) 表示额外的库代码行数。? 表示该程序应该 受到支持，但我们没有示例； ∗ 表示应该支持该程序的 2方版本（而非*N*方版本）；空白表示无法用该语言编写该程序。使用 wc -l 统计非空白、非注释行数。

程序	#	功能	Symphony	Obliv-C	Wysteria	描述
汉明距离 [14, 40]	2		16	44	?	求两个字符串的汉明距离。
编辑距离 [39]	2		46	73	?	通过动态编程求两个字符串的编辑距离，编程，求两个字符串的编辑距离。
生物匹配 [7]	2		29	73	?	计算最小欧几里得距离 介于一组点（来自 *A*）与一个单独的点（来自 *B*）在二维空间中。
数据库分析 [7]	2		94	99	?	计算以下数据的均值和方差 两个数据库的并集与连接。
最大公约数 [21]	2		10 (+4)	39	22	使用欧几里得算法求两个数的最大公约数。
最富者 [28]	*N*		6 (+7)		34	*N*方百万富翁问题的变体。
GPS [28]	*N*		21 (+11)		42	计算一维最近邻 每个*N* 参与方的邻居。
拍卖 [28]	*N*		13 (+4)		42	执行第二价格拍卖，向所有人揭示 第二高出价，并向 拍卖师揭示最高出价者。
中位数 [28]	2	R	16 (+3)	?	?	计算一个数字集合的混合模式（响应式）中位数。
交集 [28]	2	R	12 (+9)	?	?	两个集合上的朴素私有集合求交。
mmm	*N*	P,$,R	29 (+5)			使用基于比较的单败淘汰赛 在 *N* 个参与方中找出最富者。
委员会	*N*	P,D,$	26 (+45)			选举一个规模为*K* <*N*的小型委员会；适用——于在*N* 个参与方中公平委托给*K* 个参与方。
waksman [34, 13]	*N*	$	167 (+67)	∗		安全地洗牌一个数组，使用*N* 轮迭——代的 Waksman置换网络。
LWZ [23]	*N*	P,$,S	20 (+76)			使用*N* 个重新共享的份额安全地打乱数组 线性秘密共享方案（LSSS）的。
trivial-doram [15, 6]	*N*		23	∗	?	一个适用于不经意随机存取存储器的库，改编自用于多方计算的简单客户端-服务器不经意随机存取存储器。
shuffle-qs [17](清单 4)	*N*	P,D,$,S,R	54 （+76）			使用先洗牌后排序进行安全排序，采用 LWZ 作为底层安全洗牌协议，以及 快速排序作为排序算法。

MOTION 增加了支持使用现有 TCP 连接创建 Party 对象的功能（而不是由 MOTION 自行创建）。这样可以确保每对 Symphony 参与方之间只建立一次 TCP 连接，而不必像以前那样每当创建新的 Party 对象时（即每次解密时），都由 MOTION 反复建立连接。

8 实验评估

本节通过一系列实验和案例研究表明，与以往系统相比，Symphony 在编程表达能力和易用性方面更胜一筹，同时保持了有竞争力的性能。

8.1 表达能力与易用性

我们根据实现 MPC 文献中 16 个程序的经验，讨论 Symphony 在表达能力和易用性方面的优势。这些程序列于表 2。作为比较，我们考察这些程序是否能在 Wysteria [28, 30] 和/或 Obliv-C [39],中实现，以及它们是*N*方还是2方程序。我们在反应式多方计算、同步随机性和委托（均在第 2 节的表 1 中介绍），以及一等参与方集合和重新分享（在第 3.2 节中介绍）这些程序所需的 Symphony 语言特性归类于**特性** 列中。

表达能力 在这十六个程序中，我们认为 Obliv-C 能表达其中 9 个，Wysteria 能表达其中 11 个。这是因为这两种语言都缺少一些必需的功能。

Obliv-C 仅支持两个参与方，因此无法表达本质上涉及*N*个参与方的程序。正如第 2.2 节所述，Obliv-C 对协调的支持也不够完善；而 mmm、committee、lwz 和 shuffle-qs 需要对一等参与方集合执行安全计算，这些集合会根据仅在运行时才可用的值动态计算得出。mmm 和 committee 这两个基准测试均由我们设计，其中包含用于实现基于对手结构 [19]的优化技术的协调模式。

Wysteria 不支持一等参与方集合、委托、重新分享或同步随机性。Wysteria 支持参与方集合值，但只提供创建这些值的表达式形式，不支持用它们进行计算。因此，例如，无法像 LWZ 中那样动态选择参与方子集来洗牌输入。我们将在附录 A.3 中进一步讨论这一重大限制。这些限制导致 mmm、committee、waksman、lwz 和 shuffle-qs 均无法表达。为 Wysteria 添加同步随机性支持并不困难，这样一来，Wysteria 就能表达 waksman。其他程序仍然存在障碍：它们都需要一等参与方集合，而 committee、lwz 和 shuffle-qs 还需要委托或重新分享。

易用性 即使程序也可以用另一种语言表达，Symphony 仍能带来易用性方面的优势。我们以 Wysteria 作为比较对象，因为它也具备协调功能。

与 Symphony类似，如第 2.2 节所述，Wysteria 通过 let 绑定 `x =并行(P)= e` 指定哪些参与方应在给定的词法作用域中执行，从而协调各参与方；并通过 let 绑定 `x =sec(P)= e` 指定多方计算应在何处进行。然而，Wysteria 不允许在 `sec` 作用域中进行递归函数调用，这使得需要有界递归的安全算法难以编写且效率低下。

清单 1 用 Symphony 编写的表 2 中的最大公约数程序。

```
1 定义 brec gcdr (a, b) =2 多路复用 if
(a
义
```

清单2 表 2 中以 Wysteria 编写的最大公约数程序。1 let 绑定

```
gcd
ma
(a,
ma
sec
sb
```

请考虑列表 1 和 2 中的 gcd 函数。该程序分别在 Symphony 和 Wysteria 中计算 {A,B} 中两个加密整数的最大公约数。得益于灵活的第一类共享，Sy mphony 实现符合惯用写法且效率很高。相比之下，Wysteria 的实现需要多个安全区块，因为在调用 gcdr 之前（第 8 行的 b % a）和之后（第 12 行的 if (a == 0)）都要进行安全计算。请注意，第 8 行不存在除以零的风险——加密运算 b % a 是全定义的，并且当 a 为 0 时其值为 b。gcd 函数对加密值进行递归，因此我们必须使用一个公开上界（此处为 93）限制递归，以便将其表示为电路。Symphony 提供了特殊的 brec 关键字来简化这一过程：def brec gcdr a b 会展开为 def gcdr gcdr a b，其中 gcdr 参数遮蔽了同名的递归绑定。调用 unroll gcdr (const 0) 93 会获取 gcdr，并通过自应用 93 次将其“展开”，同时使用 const 0 作为基例。

由于 Wysteria 不允许在 安全作用域内进行函数调用，我们只能在对 gcdr 的每次递归调用前后进入和退出 安全 模式（第 6、10 行）。相比之下，Symphony 对一等份额的支持，使得在多方计算中能够以更符合惯用写法的方式表达 GCD 这类自然递归算法。

8.2 性能

Symphony 的表达能力并未给实现带来不必要的负担，使其难以实现良好的性能。我们将 Symphony 的性能与高度优化的多方计算框架 Obliv-C 进行了比较。实验详情见附录 A.4 和 A.5；这里我们总结实验结果。

我们在表 2 中列出的前五个程序上进行了测试。这些程序在文献中广为人知，也经常被引用。我们将 Symphony 和 Obliv-C 都配置为使用 EMP 的 [36] 2方混淆电路实现，以便将语言开销与密码学成本区分开来。在多方计算的模拟局域网上，Symphony 的运行时间是 Obliv-C 的1.15× 。不使用多方计算时，Symphony 的运行时间是 Obliv-C 的 2.4× 。在广域网上（带宽限制为 100 gbps，往返时延为 50 ms），Symphony 的运行时间是 Obliv-C 的0.85× 。分析这些开销后，我们发现其中一个来源是 Symphony 支持

$N > 2$ 个参与方：参与方纳入检查需要使用集合（通过平衡二叉树实现），而不是简单的相等性测试。更显著的开销与 Symphony 本身无关：该语言是用 Haskell 实现的解释器，而 Obliv-C 则嵌入在编译型 C 语言中。事实上，这两个框架生成的混淆电路大小非常相近。因此，我们预计采用更接近产品级质量的实现后，两者之间的差距会缩小。

Symphony 的表达能力还允许程序员直接、简便地编写算法级优化。例如，中位数和 shuffle-qs 程序利用反应式多方计算，在明文状态下执行某些比较操作，相较于单体协议，显著提升了性能 [22, 29]。尽管有些语言支持反应式多方计算，但要表达 lwz 安全洗牌所需的特性组合只有 Symphony 才具备——其他语言都无法表达这一算法。与另一种安全洗牌算法 Waksman 相比，lwz 的性能优势显著，因为它无需进行密码学计算，只需重新分享，并在明文状态下进行比较和洗牌。因此，对于所有输入规模，它的运行时间都大幅缩短。

9 结论

我们介绍了 Symphony，这是一种对协调提供强大支持的新型多方计算语言。Symphony 的作用域并行 代码块、一等参与方集合和一等份额（支持反应式计算、委托和重新分享）带来了无与伦比的表达能力；结合 Symphony 的广义 SIMD 语义，使协调编程不易出错。我们形式化定义了 Symphony 的核心，并证明程序直观的单线程解释与其实际的分布式语义一致。我们的原型实现性能可与现有系统相媲美，同时还能独特地支持各种优化。

A 实现、实验与案例研究

A.1 实现

本节提供有关 Symphony Haskell 解释器的更多信息（扩展第 7 节）。我们重点介绍三个有趣的特性：代数类型上的多方计算、同步随机性以及标准库。

代数类型上的多方计算 Symphony 将 λ-Symphony的 共享、多路复用、`case`和揭示进行了推广，允许将任意代数类型作为参数。它还增加了另一个表达式 `mux-case`，用于对加密（共享）值进行分类讨论。对对使用 共享、`mux-case` 和 揭示表达式通常并不令人意外。对乘积（对）进行共享，是通过递归地逐分量共享来实现的。

$$\text{共享}[\phi, \tau_1 \times \tau_2 : P \to Q]\,(a, b) = (\text{共享}[\phi, \tau_1 : P \to Q]\,a, \text{共享}[\phi, \tau_2 : P \to Q]\,b)$$

对加密和类型执行 mux-case 操作与 case 相同，而揭示操作的工作方式与共享类似。

我们将和类型（变体）表示为带标签的对。 λ-Symphony 值 $\iota_0\, v$ 表示为和类型〈true, v,默认值〉，而值 $\iota_1\, v$ 表示为和类型〈false,默认值, v〉。默认值是一个占位符；如果共享和类型值，它将被替换为适当类型的默认值。与对一样，共享和类型值的实现方式是递归地逐分量共享。

共享$[\phi, \tau_1 + \tau_2 : P \to Q]$和类型$\langle b, v_1, v_2\rangle$
$\equiv$和类型$\langle$共享$[\phi, \mathbb{B} : P \to Q]\, b$, 共享$[\phi, \tau_1 : P \to Q]\, v_1$,共享$[\phi, \tau_2 : P \to Q]\, v_2\rangle$

要共享默认值，share$[\phi, \tau : P \to Q]$默认值时，我们只需改为共享适当类型 τ 的默认值。对于 mux-case，类型τ 必须是幺半群，且默认值必须是单位元。例如，当 τ 为布尔值时，单位元是 false，幺半群运算为异或。同样，当 τ 为自然数类型时，单位元是0，运算为 +。对和类型值执行揭示操作的方式类似。

mux-case sum〈b, v_0, v_1〉 $\{\theta_1 \to e_1; \ldots; \theta_n \to e_n\}$ 表达式是我们需要考虑的最有趣的一种。标签 b 已加密，因此我们无法检查它来判断它是否与左注入模式或右注入模式匹配。那么，我们该怎么办？mux-case 首先会筛除所有既非左注入模式也非右注入模式的模式 θ_1、……、θ_n。然后，它会遍历剩余的模式 $\theta_k \equiv \iota_i$x 5，并生成如下结果：当 i 为 0 时，mux b then default else v'_0；当 i 为 1 时，mux b then v'_1 else default。其中，v′ i 是在环境 (γ) 扩展了 [x $\mapsto v_i$] 后，对 e^k 求值得到的结果。此时，我们将刚才生成的值列表 $v_1, \ldots, v_m$ 相加（其中 m $\leq$n 对应于 θ_1、……、θ_n 中属于左注入模式或右注入模式的模式数量）。加法过程是与这些值 v_1,……v_m 的类型 τ 关联的幺半群运算。

考虑将布尔值标准编码为单位类型之和：bool := unit + unit。我们将 sum〈true, •, •〉视为 true 的编码，false 也对称地处理。我们希望用 mux-case 表示 mux 的显而易见的编码能够奏效：mux b then e_1 else e_2 := mux-case sum〈b, •, •〉 $\{\iota_{\texttt{true}}\, \bullet \to e_1;\ \iota_{\texttt{false}}\, \bullet \to e_2\}$。确实如此。在下面的计算中，令 v_i 表示计算 e_i所得的结果。令 τ 为 v_1 的类型，并令 v_2, i d_τ 表示 τ 上的幺半群单位元，add$_\tau$ 表示幺半群运算。那么，我们有：

mux-case sum〈true, •, •〉 $\{\iota_0\, \bullet \to e_1; \iota_1\, \bullet \to$
$e_2\}\equiv$add$_\tau$ (mux true then v_1 else id$_\tau$)(
mux true then id$_\tau$ else v_2)$\equiv$add$_\tau$ v_1 id$_\tau\equiv v_1$

代数类型上的多方计算是一个有趣的问题，而这一领域还有许多尚未解决的问题。我们建议感兴趣的读者参阅 Ye 和 Delaware [38] 撰写的《Oblivious Algebraic Data Types》，以获得更完整的介绍。

[5]为简洁起见，我们忽略嵌套模式。这里描述的过程很容易扩展。

同步随机性 Symphony 提供了一种便捷的方法，可在多个参与方之间并行生成随机性，确保所有参与方都获得相同的随机值。如下方的 Symphony 代码所示，只需访问本地随机性，就可以将此功能实现为一个库。

```
1 def randomSend P n  =2 let rec sum' = fun Q ->
3 c
> I
let
```

此代码会在P 中的每个参与方生成一个随机值，并将它们全部相加，再对n取模，从而生成一个范围为$0..n-1$的随机自然数。这种方法可行，但效率低下，因为每当参与方希望生成同步随机数时，都需要彼此通信。6

由 Symphony 提供的原语表达式 `rand P` μ（其中 μ 是类似 自然数类型 的基类型），提供了相同的功能，但效率更高。它会在参与方P 首次请求同步随机数时，使用上述过程在各参与方之间建立一个共享种子。此后，参与方会使用本地的密码学安全伪随机数生成器，无需通信。

标准库 Symphony 随附的标准库包含了函数式编程语言中常见的许多组件，例如用于选项的库；针对各种代数类型（自然数、选项、列表、映射等）的消去子（折叠）；以及高阶函数（flip、compose、curry、uncurry 等）。不过，它也包含一些不寻常的组件，例如用于协调、有界递归和同步随机性的工具库。接下来，我们将介绍一些有趣且具有代表性的函数。

[6]读者可能会注意到，此过程生成的随机数略有偏差。当我们使用此过程来生成种子（如下所述）时，结果是无偏的，因为该种子是机器字长的倍数（实际中为128位）。

清单3 Symphony 标准库中的部分函数。 1 def bundleUpWith f P =
cas
bu
f (u
rar
mi
15
1n
shu
24

协调库主要处理主体集合、捆绑包及其相互作用中的常见操作。例如，清单 3 中的 `bundleUpWith` 函数接收一个函数`f` 和一个参与方集合`P`，并通过在`P`中的每个参与方`p` 本地调用`f` ，在`P`中的参与方之间创建一个捆绑包。此函数可用于消除从每个参与方读取输入这一常见模式所带来的样板代码。

```
def readInputs P = bundleUpWith (fun _ -> read nat) P
```

Symphony 中的有界递归库提供了一个关键函数`unroll`，该函数在清单 1 中有所使用并作了简要介绍。 `unroll` 的完整定义见清单 3。如第 8.1 节所述，`unroll f init n` 函数与`brec` 关键字结合使用，可在最终通过调用`init`结束之前，将函数`f`有限地“展开”`n` 次。为说明这一点，请看第 8.1 节中的最大公约数和`gcdr`函数。

```
1 defbrec gcdr (a, b)  = 多路复用 if (a  == 0) then b else gcdr ((b % a),
a)
```

签名 定义 brec $fx_1 \ldots x_n$ 会被降糖为签名 定义 $ffx_1 \ldots x_n$，这样一来，f 函数体中的任何递归调用现在都会引用形式参数f ，而不再引用正在定义的函数。通过遮蔽递归绑定，我们可以透明地将任何递归函数转换为可用于有界递归的函数。以下是`gcdr`的示例：

Symphony：使用协调的表达性安全多方计算

```
定义gcdr gcdr（a，b） = 多路复用 如果（a  == 0）那么 b 否则 gcdr（（b % a），a）
```

最后，我们可以计算 0 和 2 的有限展开，以查看 展开 如何与使用`brec`进行语法糖消解的函数结合使用。

```
1 def gcd = 展开 gcdr (const 0) 0 2                = if  0 == 0  then (const 0) else gcdr (展
2
3                =gcdr (展开 gcdr (const 0) 1) 4                = gcdr (if  1 == 0  then (const 0)
0)
```

外部函数接口与资源管理 Symphony 解释器使用 Symphony 运行时库实现多方计算。Symphony 运行时库使用 EMP 和 MOTION 库实现多方计算。这些层（Symphony 解释器、Symphony 运行时以及 EMP/MOTION）通过外部函数接口进行交互，并使用不透明的外部指针来表示下一层中的值。

在 Haskell 中，原始外部指针由 ForeignPtr 类型封装；当其被垃圾回收时，该类型会在原始外部指针上执行关联的 终结器 （即一个 Haskell 函数）。我们利用此终结器调用（通过外部函数接口）适当的析构函数，该函数由 Symphony 运行时提供。

在 Rust 中，原始外部指针由 newtype 风格的 Rust 结构体封装。这些 Rust 结构体包含 Drop 特征的显式实现，并在结构体被丢弃时调用其 drop 函数。这类似于 Haskell 中的 ForeignPtr，不同之处在于，Rust 可以静态插入对 drop 的调用，而不必依赖垃圾回收。与 Haskell 中一样，drop 函数会调用（通过外部函数接口）适当的析构函数，该函数由 EMP 和 MOTION 提供。

最后，我们为 EMP 和 MOTION 实现外部函数接口，这些接口提供执行堆分配和释放的构造函数与析构函数，用于处理必要的 C++对象。

综上，当一个加密值被 Haskell 垃圾回收时，ForeignPtr 将调用 Symphony 运行时定义的相应析构函数。随后，该析构函数会丢弃该值，从而调用 EMP 或 MOTION 定义的相应析构函数。最后，EMP 或 MOTION 通过外部函数接口公开的析构函数将使用 delete 关键字释放 C++ 对象。

这些将以不同语言编写的软件集成起来的方法大多已是标准做法。我们选择将对 Symphony 运行时的增强实现为一个独立的库，以优化性能。我们特意选择 Rust，因为它拥有出色的库、工具链和文档。Symphony 运行时采用惯用 Rust 编写，因此也可能成为具有独立研究价值的成果，供

需要使用支持委托、重新分享和反应式多方计算的多方计算协议的研究人员使用。

A.2 基准程序

这里我们将更详细地介绍在 Symphony 中实现的部分程序，这些程序列于表 2 中。第 8.1 节简要讨论了实现各程序所需的语言特性。

此处列出的所有程序以及用于基准测试的模板，其实现均可在 Symphony代码库中找到：https://github.com/plum-umd/symphony-lang。

汉明距离、**编辑距离**、**生物匹配**、**数据库分析**和**最大公约数**是双方程序，已在表 2 中完整描述。我们使用 Symphony 和 Obliv-C 实现了这些程序；Obliv-C 仅支持双方多方计算。我们相信这些程序也可以用 Wysteria 实现。

最富者、**GPS**、**拍卖**、**中位数**和**交集**也已在表 2 中完整描述。它们先前已在 Wysteria 中实现，我们将其回移植到了 Symphony。**最富者**、**GPS**和**拍卖**支持N个参与方，但除此之外不需要特殊协调。**中位数**和**交集**支持2 个参与方，并使用 *ReactiveMPC*（Wysteria 称之为“混合模式”）来提高效率；我们相信这些程序可以用 Obliv-C 实现。

mmm 在 N 个参与方之间组织一场单败淘汰赛。比赛中的每一场对决都是一次由 2 名参与方执行的安全计算，获胜者晋级下一轮。该程序需要 一等参与方集合，因为每场对决的参与者都要从剩余选手集合中动态分配。同样，剩余选手集合也要根据上一轮对决的结果动态确定。程序还需要 同步随机性 来确定每场对决的参与者。最后，程序需要 反应式多方计算，以解密每轮的获胜者，从而协调他们参加下一轮。

委员会 通过选出一个大小为 $K < N$ 的随机委员会，并将安全计算委托给该委员会，从而在 N 个参与方之间执行任意安全计算。该程序需要一等参与方集合和同步随机性，因为委员会是根据 K 以及由 N 个参与方生成的同步随机数动态计算得出的。程序还需要委托，因为委员会之外的参与方会加密其输入并发送给委员会，但不会参与安全计算。

Waksman 协议可在不向任何参与方揭示排列的情况下，在N 个参与方之间安全地洗牌数组中的元素。Waksman协议通过反复应用经典的Waksman置换网络 [34] 来实现安全洗牌。置换网络会反复对列表中的两个元素进行条件交换，直到列表完全打乱；每次交换都可以通过布尔门实现。置换网络的一个微妙之处在于，必须选择网络的控制位，该输入决定哪些交换门确实应交换其输入

。为了对所有参与方完全隐藏洗牌过程，每个参与方都会秘密选择自己的控制位，并对序列中的一个$|P|$ 网络进行编程。因此，完整的洗牌需要 $|P|$ 个网络，而实现还需要进行一些协调：程序员必须规定每个参与方依次对一个网络进行编程。该协议需要同步随机性来执行非安全洗牌这一子程序。

LWZ 也是一种安全洗牌协议，最初由 Laur、Willemson 和 Zhang [23]提出。他们展示了如何利用线性秘密共享方案，在Q个参与方之间实现高效的安全洗牌，并抵御T 次腐化。LWZ 协议通过在委员会之间反复洗牌元素来实现安全洗牌，这些委员会是参与方集合的真子集，大小为 $|Q|-T$。委员会获得输入列表的份额，并就一个随机排列达成一致， π；随后根据 π 在本地对其份额进行排列；然后为下一个委员会构造新的份额。此过程不断重复，直到每组T 个参与方都曾被排除在某个委员会之外，这足以隐藏全局洗牌。[7] 除同步随机性外，**LWZ** 还需要一等参与方集合和重新分享，以计算所有大小为$N-T$ 的参与方集合，并将元素数组重新分享至这些子集及从这些子集重新分享出来。据我们所知，此前没有任何多方计算语言能够直接表达**LWZ**。

朴素 ORAM Trivial ORAM [35] 允许将秘密份额用作地址，从而模拟随机访问。每次访问（读取或写入）时，模拟过程都会线性扫描底层数组。使用多路复用器选择要操作的索引。TrivialORAM 是一种基础原语，可用于构建更高效的 ORAM 实现，例如 Circuit ORAM。

[7] 对于安全性而言， $|Q|-T>T$（等价地， $|Q|\geq 2T+1$）这一条件也是必要的；否则，一个由 $|Q|-T$ 人组成的恶意委员会就能串通，揭示正在洗牌的秘密元素。

清单 4 以 Symphony 编写的安全*N*方排序过程。采用"洗牌后排序"范式 [17]，并以 LWZ 作为底层洗牌算法。{*A*,*B*,*C*, *D*, *E*} 中的每个参与方都提供一个整数数组，这些数组连接在一起后，由 Q 中的参与方以安全方式进行洗牌和排序。

```
1  参与方 AB CD E
2
3  -- read input at p, secret-share to all in Q
4  定义readShare Q p  = 并行 ( p ) \/ Q)
5    let 绑定 i  = 并行 { p } 读取 (数组 整数) 来自 "lwz.txt" 在 -- file local to
     ↪ each p
6    共享 [gmw, 数组 整数 : { p } -> Q] i
7
8  定义委托份额 P Q  =
9    映射 (读取 readShare Q) (psetToList P)
10
11 定义 使用.....打乱 Q S Q 的份额 = 并行 (Q \/ S)
12   let 绑定sharesS = 共享 [gmw,数组 int : Q -> S] Q 的份额 in
13   共享 [gmw,数组 int : S-> Q] (洗牌 S sharesS)
14
15 定义 lwz Q Q 的份额  =
16   let 绑定t  = 1 在
17   foldr (使用 Q 打乱) Q 的份额 (子集 Q ((幂集大小 Q) - t))
18
19 定义揭示小于等于 Q x y = 揭示 [gmw, 布尔值 : Q -> Q] x  <= y
20
21 定义 安全排序 Q 份额列表 =
22   let 绑定 Q 的份额   = 并行 Q (数组连接 份额列表) 在
23   let 绑定已打乱序列 = lwz Q 的份额 在
24   let 绑定 已排序    = 快速排序 (揭示小于等于 Q) 已打乱序列
25
26 定义 main () = 并行 {A,B,C,D,E}
27   let 绑定Q  ={A,B,C} 在
28   let 绑定份额列表  = 委托份额 {A,B,C,D,E} Q 在
29   let 绑定已排序      = 安全排序 Q 份额列表 在
30   ...
```

shuffle-qs 是最复杂的程序，它将**委员会** 和**lwz** 都用作子程序。它通过选取一个由 $K > 2$名参与方组成的委员会，使用**lwz**在委员会成员之间洗牌，然后通过揭示每次比较的结果，使用快速排序对元素进行排序，从而实现对N 个参与方的安全排序。

清单 4 展示了用 Symphony 编写的**shuffle-qs** 程序。按照惯例，所有参与方都运行同一程序，从`main`开始。虽然洗牌和排序代码适用于任意数量的参与方，但`main` 针对顶部声明的那些参与方进行了专门处理，它们的名称为 A–E。当执行到第 26 行的并行 {A,B,C,D,E} ... 时，只有列出的参与方会执行后续代码 ...。第 27 行将 Q 绑定为集合{A,B,C}，随后在第 28 行将其作为第二个参数传递给委托份额，并在第 29 行将其作为第一个参数传递给安全排序 。

在readShare Q p中，参与方p 从本地文件 lwz.txt（第 5 行）读取一个整数数组，然后在Q 中的参与方之间创建共享（在 Symphony 中，数组的共享是一个份额数组）。共享 操作要求{p}∪ Q 中的所有参与方都在场

（由第 4 行的并行 确保），这样P 就能向Q 中的每个参与方发送自己的份额，并确认它们已准备好接收——请注意，p 可能是Q的成员，也可能不是。delegateShares 函数会为p ∈ P中的每个参与方调用readShare Q ，目的是委托后续计算给Q中的那些参与方。

在第 12–13 行，参与方 Q 之间的份额会重新分享给参与方 S，随后经 S洗牌后，再重新分享回 Q 。第 17 行的 foldr 会对 Q 的每个子集 S （由 subsets计算得出）调用 shuffleWith，该子集的大小为 $|Q| - T$。随后，此函数会在这些参与方 S之间重新分享sharesQ ；这些参与方调用 shuffle 来排列其中的值，然后再将结果重新分享回来。在shuffle中，参与方 S 会共同确定一个伪随机数生成器的种子，并以此为基础进行洗牌，从而确保它们计算出相同的排列。Laur、Willemson 和 Zhang 证明，如果每个子集S的大小为$|P| - T$，那么除非$n > T$ 个参与方串通，否则无法获知已打乱元素的顺序。在清单 4 中，我们在第 16 行指定了$T = 1$ 。

第 23 行执行完毕后，shuffled 中包含一个经过洗牌的秘密共享数组。接着，第 24 行使用revealLte Q 作为比较函数，对该数组执行quicksort。此函-数使用 Symphony 的reveal 结构，将P 之间的份额转换为在Q之间复制的明文值。

该程序依赖于一等参与方集合、委托、同步随机性和重新分享，因为它使用了**委员会** 和**lwz**。由于快速排序中的每次比较都会先被解密，然后才能安全地计算下一次比较，因此它需要反应式多方计算。

A.3 Wysteria 中的委员会选举

由于缺乏对参与方集合进行计算的支持，Wysteria 不支持一等参与方集合。我们可以通过尝试将清单 5 中 Symphony 的选举 函数移植到 Wysteria 来看到这一点，但最终证明这是不可能的。这一限制影响了 Wysteria 以组合方式表达协调协议的能力。

■ **清单 5** 一个简单的 Symphony 函数，用于从 P 中选举出 $<= k$ 个参与方的子集。委员会程序使用了该函数的更复杂版本（见表 2）。

```
1 定义 选举 Pk  =2 如果 k == 0则 { }3     否则情况P4 {
{}
1)
```

elect P k 函数从集合 P 中“选举”出至多包含 k 个参与方的子集。更完善的实现可能会使用投票程序或同步随机性来确定该子集，但这个简单版本已足以满足我们的需求。下面来看看如何用 Wysteria 表达这个函数。

尝试1 表达选举 最自然的方式，是使用与 Symphony 相匹配的签名。

```
let elect (P : ps{true}) (k : nat) : ps{subeq P} = ??
```

这里我们可以看到一种相当典型的类 OCaml 签名，其中包含一些类型注解。Wysteria 的类型系统是精化类型系统，但经过限制，精化只能放置在参与方集合类型 `ps`上。精化集合包含标准的命题精化 `true` 和合取（与），以及特殊精化 `singl`、`subeq x` 和 `eq x`。`singl` 精化由单例集合满足，这些集合记为 P：$|P| = 1$。`subeq Q`（`eq Q`）精化由参与方集合P满足，表示它们是Q的子集（或与Q相等）：$P \subseteq Q$。

在这个 选举 的签名中，我们看到 `P` 的类型为 `ps{true}`，这表明它是一个未精化的参与方集合。此外，我们看到 选举 应返回一个属于 `P` 子集的参与方集合。将 `P` 视为返回类型中的绑定变量，是依赖类型系统和精化类型系统的一个标志。

遗憾的是，这次尝试一开始就行不通。我们无法实现这个签名，因为 Wysteria 根本没有与消去（即对参与方集合进行计算）相对应的表达式形式。在下一次尝试中，我们将尝试通过把参与方集合编码为数组，并改为对数组进行计算，来绕过这一限制。

尝试 2 在这次尝试中，我们将把参与方集合 `P` 编码为一个数组，其中包含属于`P` 的元素所对应的单例集合，其类型为 数组 参与方集合{单例且是 P 的子集}。这种方法显示出更大的希望：

```
1 let 选举 (P : 参与方集合{true}) 2 (Pe: 数组参与方集合
{单
{单
Pe
ret
in
```

此函数遍历Pe 的前k 个元素，并将它们赋给一个新数组ret，然后返回ret。与我们上次的尝试不同，参数P 在此签名中仅用于细化Pe 的类型和返回类型。所有计算都在参与方集合的编码上进行。

这个版本的 选举 类型正确，因此我们可以尝试在一个简单示例中使用它。

```
1 let foo(Q :ps{true}) (Qe : 数组 ps{singl 与 subeq Q}) : ... = ... 2 3 let P = { !
A, !
in (
=9
```

这个小型驱动程序声明P 包含参与方A、B和C ，它们被编码在Pe中。它使用上面的选举函数生成一个编码后的委员会Ce，其中包含2 个参与方。此时，我们只想让委员会成员运行foo，但要做到这一点，我们必须用实际的参与方集合填入空缺（??），该集合由编码Ce表示。为此，我们定义一个解码 函数。

```
1 let 绑定 decode (P : ps{true}) 2 (Pe: array
ps{
len
nth
(ac
```

decode 函数会遍历Pe ，并对其所有元素取并集。由于这些元素都被标记为P的子集，Wysteria 能够推断，所有元素的并集同样是P 的子集，如返回类型ps{subeq P}所示。8 让我们试着用decode 函数填上前面示例中的那些空缺。

```
1 let 绑定 foo(Q :ps{true}) (Qe : array ps{singl 与 subeq Q}) : ...  =  ... 2 3 let 绑
定
Pe
8 le
```

现在，我们可以进入一个只包含参与方 C 、使用 par(C)模式的执行上下文。遗憾的是，类型系统在调用foo时拒绝了这个程序。为什么？因为 Wysteria 无法推断出 Ce 的类型是array ps{singl and subeq C}，而这是foo的签名所要求的。相反，Ce 的类型是array ps{singl and subeq P}。我们可以设想为foo使用另一种签名，但对于组合性而言，foo 接收一个编码的参与方集合这一点很重要。这样一来，foo 就能利用其他任何根据我们的编码动态计算参与方集合的函数。例如，foo 可能希望对其参数调用elect 。

当然，关键问题在于选举 的返回类型不够精确，因此选举 返回的编码参与方集合无法传递给其他需要对参与方集合进行计算的函数。此时我们就卡住了。

[8]Wysteria 不支持我们用于计算 n 的数组长度原语。添加这一功能很容易，而且可以避免在签名中加入数组长度，令其变得冗长。

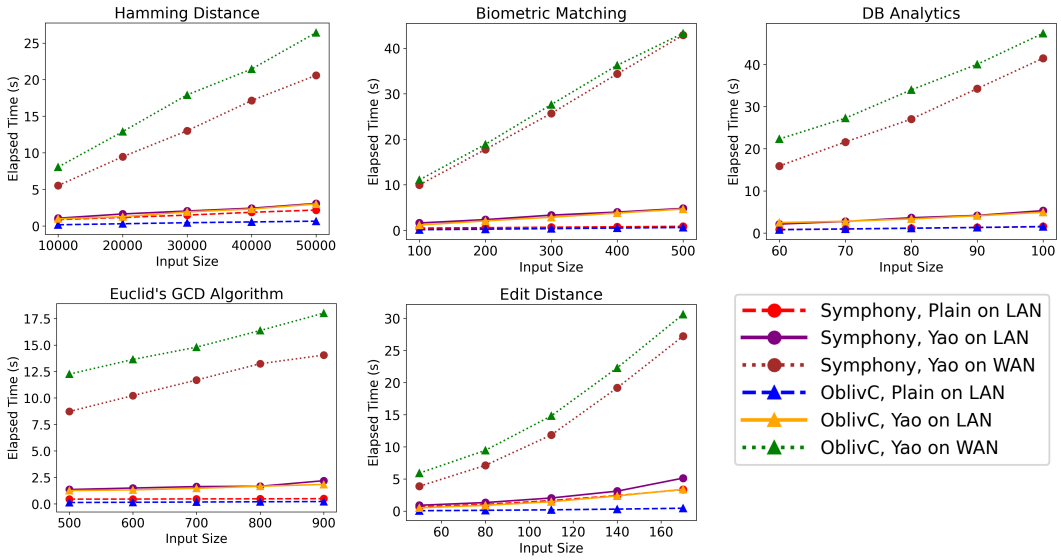


图 7 5 个程序的端到端执行时间，取五次样本的平均值（越低越好）。**局域网** 是模拟的 1 Gbps 连接，无延迟。**广域网** 是模拟的 100 Mbps 连接，往返时延为 50 ms。**Yao** 和 **Plain** 协议分别使用 EMP 的 sh2pc（半诚实、两方）协议和 Plain 协议。Symphony 使用 EMP 的整数接口。Obliv-C 使用 EMP 的位接口（将整数运算编译为电路）。除 gcd-gc 外，所有基准测试的输入规模均表示作为输入提供的列表长度；对于 gcd-gc，输入规模表示最大公约数算法的迭代次数。

Wysteria 没有提供其他机制来规避参与方集合缺少消去形式的问题。

A.4 与 Obliv-C 的性能比较

我们在相同的程序上，将 Symphony 的运行时间和门计数方面的性能与 Obliv-C 进行了比较。结果总结于第 8.2 节；本节将详细介绍实验和结果。

实验设置 为了公平比较，我们将 Symphony 和 Obliv-C 都配置为使用 EMP [36] 作为其多方计算后端；我们扩展了 Obliv-C，使其通过回调接口使用 EMP。我们使用 Symphony 和 Obliv-C 实现了包含五个程序的基准测试套件：汉明距离、编辑距离、生物匹配、数据库分析和最大公约数。有关这些程序的说明，请参见表 2。

实验在一台 2019 款 MacBook Pro 上进行，其配备 2.8 GHz 四核 Intel Core i7 处理器和 16 GB 内存（OSX 11.3.1）。Obliv-C 编译器是 GCC 5.5.0 的扩展，所有基准测试均使用 -O3 优化进行编译。实验在两个模拟网络中进行：局域网（带宽为 1 gbps，往返时延为 <1ms）和广域网（带宽为 100Mbps，往返时延为 50 ms）。除 gcd-gc 使用 64 位整数外，所有实验均使用 32 位整数。报告的执行时间是参与方A的端到端执行时间，取五次测量的均值。

运行时间 图 7 绘出了 Symphony 和 Obliv-C 在各项基准测试中的端到端执行时间。在局域网的 MPC（Yao）场景下，Symphony 的运行时间是 Obliv-C 的1.15× 倍（按几何均值计算）。不使用 MPC 时，Symphony 的执行时间是 Obliv-C 的2.4× 倍。在广域网的 MPC 场景下，Symphony 的执行时间是 Obliv-C 的0.85× 倍。最大减速出现在 edit-dist 中；该基准使用动态规划，而 GCC 对 Obliv-C 的这一实现做了大量优化。

Symphony 的开销主要有两个来源：首先，Symphony 支持任意数量的参与方，而 Obliv-C 仅支持两个。这一点意义重大，因为 Symphony 会频繁对作用域内的参与方进行运行时检查。由于 Symphony 支持任意数量的潜在参与方，我们将作用域内的参与方表示为一个集合（通过平衡树数据结构实现）。因此，对主体的检查是通过集合操作实现的。我们可以使用位集而非平衡树来实现这些运行时检查，从而提高其效率。Obliv-C 也会对参与方执行某些检查，但由于它仅支持两个参与方，因此这些检查通过简单的整数相等性比较实现。

其次，Symphony 使用解释执行，而 Obliv-C 则经过编译。解释执行会带来开销，尤其是在涉及循环的程序中。例如，在单个参与方上执行一个简单的压力测试，将 1 百万个整数（明文）求和，结果显示 Symphony 耗时约 6 秒，而 Obliv-C 耗时约 100 毫秒。该压力测试不会执行 λ-Symphony 所引入的任何运行时检查，这表明开销源于解释执行。

由于 Symphony 和 Obliv-C 都是同步的（即从网络读取时会阻塞），因此每个非本地多方计算操作都会给实际执行时间增加一个 RTT 的延迟。如果实现采用异步方式，多方计算操作和解释执行便可并行进行。这样一来，非本地多方计算操作之间的实际执行时间就不再是延迟之和，而是解释执行时间与 RTT 中的较大值。除最快的局域网外，所有网络的 RTT 都为 $>$ 5毫秒。我们推测，对于大多数部署场景，Symphony 的解释执行开销足够小，完全会被网络延迟所掩盖。如果确实如此，异步 Symphony 与 Obliv-C 之间的实际执行时间将难以区分。

对比局域网和广域网的基准测试结果可以确认，在广域网部署多方计算时，Symphony 带来的语言开销远低于网络通信所需的时间。我们认为，Symphony 在广域网环境下比 Obliv-C 更快，是因为它采用了 EMP 整数接口；与 Obliv-C 的回调机制所使用的位接口相比，该接口能更高效地利用网络，因此也优于 Obliv-C 所采用的 EMP 后端。

生成的电路规模 作为第二项实验，我们对 EMP 后端进行了插桩，以统计所使用的与门和异或门数量。统计门数主要是一项合理性检查，用于确保 Symphony 没有错误地引入大量不必要的门。表 3 列出了 Symphony 和 Obliv-C 生成的与门和异或门数量。Symphony 和 Obliv-C 生成的门计数非常相近，差异是由于使用了 EMP 整数接口，而 Obliv-C 则编译为位接口（这是其回调机制所要求的）。

表 3 部分基准程序的门计数（与门和异或门）。**输入规模** 对于汉明距离、生物信息匹配、数据库分析和编辑距离而言，是输入列表的长度；对于最大公约数，则是最大公约数迭代次数。门计数通过修改 EMP 来记录与门或异或门的执行次数。Symphony 在适用时使用 EMP Integer 接口；OblivC 使用 EMP 位接口（将整数运算编译为电路）。

		OblivC		Symphony		Δ(OblivC - Symphony)	
基准测试	输入规模	与门	异或门	与门	异或门	与门	异或门
汉明距离	10000	1249875	3159595	950000	2550000	299875	609595
	20000	2499875	6319595	1900000	5100000	599875	1219595
	30000	3749875	9479595	2850000	7650000	899875	1829595
	40000	4999875	12639595	3800000	10200000	1199875	2439595
	50000	6249875	15799595	4750000	12750000	1499875	3049595
生物信息匹配	100	2617868	6353496	2675500	8007000	-57632	-1653504
	200	5235768	12706996	5351000	16014000	-115232	-3307004
	300	7853668	19060496	8026500	24021000	-172832	-4960504
	400	10471568	25413996	10702000	32028000	-230432	-6614004
	500	13089468	31767496	13377500	40035000	-288032	-8267504
数据库分析	60	4609304	9569553	4732457	10425422	-123153	-855869
	70	6246968	12970159	6413597	14128242	-166629	-1158083
	80	8133020	16886701	8349937	18393262	-216917	-1506561
	90	10268008	21320663	10541477	23220482	-273469	-1899819
	100	12651370	26270229	12988217	28609902	-336847	-2339673
GCD	500	2360091	6735821	2302192	6910016	57899	-174195
	600	2832991	8085621	2762592	8291916	70399	-206295
	700	3305891	9435421	3222992	9673816	82899	-238395
	800	3778791	10785221	3683392	11055716	95399	-270495
	900	4251691	12135021	4143792	12437616	107899	-302595
编辑距离	50	780882	1704539	637372	1779607	143510	-75068
	80	2003142	4378936	1631872	4556407	371270	-177471
	110	3790602	8291824	3085372	8614807	705230	-322983
	140	6143262	13443100	4997872	13954807	1145390	-511707
	170	9061122	19832759	7369372	20576407	1691750	-743648

EMP 的电路编译器与 Obliv-C 的电路编译器所执行的优化相似，但并不完全相同。

总体而言，我们的实验表明，语言设计本身不会对端到端执行时间或生成的电路大小造成显著开销。我们将更复杂的实现留待未来工作，该实现将利用编译和编译器优化。

A.5 安全洗牌的安全性与性能

Waksman 和 LWZ 协议在安全性与性能方面各有取舍。

给定一个包含n 个位宽为w的整数的输入列表，Waksman 置换网络是一种递归算法，需要$O(w \cdot n \log n)$ 个布尔门。由于必须重复使用该网络 $|P|$ 次，因此总共需要$O(|P| \cdot w \cdot n \log n)$ 个门。该网络的电路深度随$O(\log n)$增长；这一点很重要，因为 GMW 等交互式安全多方计算协议的轮次复杂度会随深度增长。总共需要$O(|P| \cdot \log n)$ 轮通信。

另一方面，LWZ 协议避免了执行通用多方计算电路求值的需求。事实上，该协议极其轻量：LWZ 协议

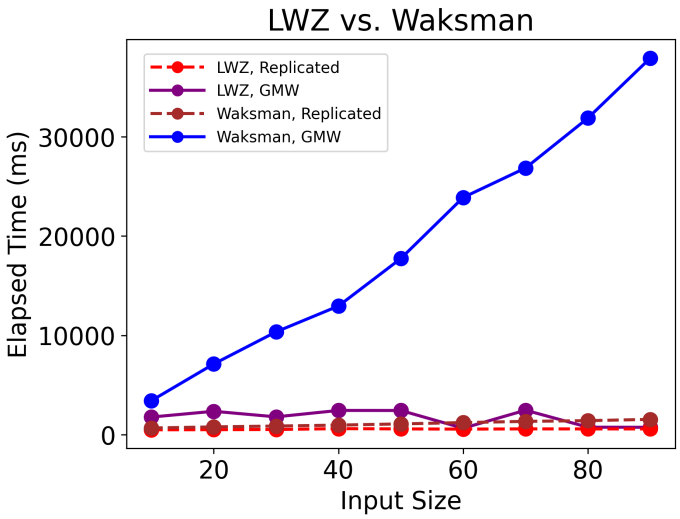


图 8 三方参与时 Waksman 洗牌与 LWZ 洗牌的端到端执行时间，取五次样本的平均值（越低越好）。**复制式**协议在不使用密码学的情况下执行程序，所有操作均以明文方式执行。**GMW**协议使用 Symphony 对 GMW 的实现，并以 MOTION 作为后端。**输入规模**表示各参与方提供的整数列表长度。

完全不需要执行任何安全门。LWZ 的缺点是，其性能会随着可容忍的腐化数量 t的增加而下降。也就是说，假设至多有t 个参与方会串通并相互共享信息。为防止这些对手获知元素的最终排列，我们必须确保对于每个由t 个参与方组成的子集，都存在一次协议重复执行，使得该子集中的任何一个参与方都不在委员会中。因此，我们必须将每个委员会的规模设为 $|P|-t$，而所需的重复执行次数会随 $\binom{|P|}{|P|-t}$ 增长。如果t 较小，例如$t = 1$，那么 LWZ 协议的性能非常出色，只需 $|P|$ 轮通信。如果t 较大，例如 $t = \lfloor |P|/2 \rfloor - 1$，那么性能会随着 $|P|$ 呈指数级下降。

—

Symphony 执行时间图 8 展示了 Symphony 执行 LWZ 洗牌和 Waksman 洗牌的端到端执行时间。在这两种协议中，三个参与方各自共享一个包含**输入规模**个整数的数组，随后将这些数组串接并进行洗牌。我们分别使用**GMW** 协议和**复制式**协议运行这些程序，并将后者作为基线。在**复制式**协议中，布尔门在本地实现并以明文方式计算，而非通过密码学实现。对于 LWZ 阈值，我们采用较为乐观的设置，即最多有$t = 1$个参与方串通。

我们的结果表明，LWZ 的 Symphony 实现恰当地避免了多方计算开销：正如前文所述，LWZ 是一种轻量级协议，因此 Symphony 不应——也确实没有——仅仅因为我们操作的是 GMW 份额就引入不必要的成本。正如预期，我们发现，基于 **GMW** 的 Waksman 实现比 **Replicated** Waksman 和 LWZ 的两种变体都慢得多。性能下降主要是由于在多方计算中执行布尔门所需的密码学操作。

我们确实注意到，基于 **GMW** 的 Waksman 性能低于预期。我们观察到，性能低下是由 MOTION 后端造成的：在此基准测试中，它会为每个参与方分配 >4 GB 内存来存储 GMW 电路。此外，执行每个门都需要访问许多不连续的内存地址，导致空间局部性较差。我们认为，在 Symphony 的兼容层中处理更多电路生成和执行工作，可以大幅提升性能。

即使采用高度优化的 GMW 后端，LWZ 协议在 $t = 1$ 的场景下仍然最为理想：对于这种场景，协调需求繁重的 LWZ 协议显然是更优的技术。Symphony 的特性使得该协议中复杂的协调过程易于表达。

B 元理论

B.1 单线程语义

图 9 展示了第 4.2 节讨论的完整单线程语义。

请注意，处理输入/输出的规则要求模式为单例参与方；这对于确保兼容性（即确保所有参与方对共享变量的内容达成一致）至关重要。

和类型与对基本上采用标准形式，只需考虑其值的位置；参与方集合通过集合并构造，并通过模式匹配分解。

Symphony 直接支持列表和数组；在 λ-Symphony 中，它们可以通过迭代的和类型值与对值进行编码，其中 nil $\triangleq \iota_1\ 0$ 和 cons $\triangleq \lambda x.\ \lambda xs.\ \iota_2\ \langle x, xs\rangle$；列表可以通过使用 case 进行模式匹配来分解。我们可以将捆绑包编码为一个关联列表，实现从参与方到位于该参与方的位置值的映射。例如，下面的列表表示一个捆绑包，其中 8 位于 A，而 3 位于 B。

$$\iota_2\ \langle\langle\{A\}, 8@\{A\}\rangle, \iota_2\ \langle\langle\{B\}, 3@\{B\}\rangle, (\iota_1\ 0)\rangle\rangle$$

（为避免冗余，列表本身缺失的位置注释已省略；它们全部都是 $@\{A, B\}$。）

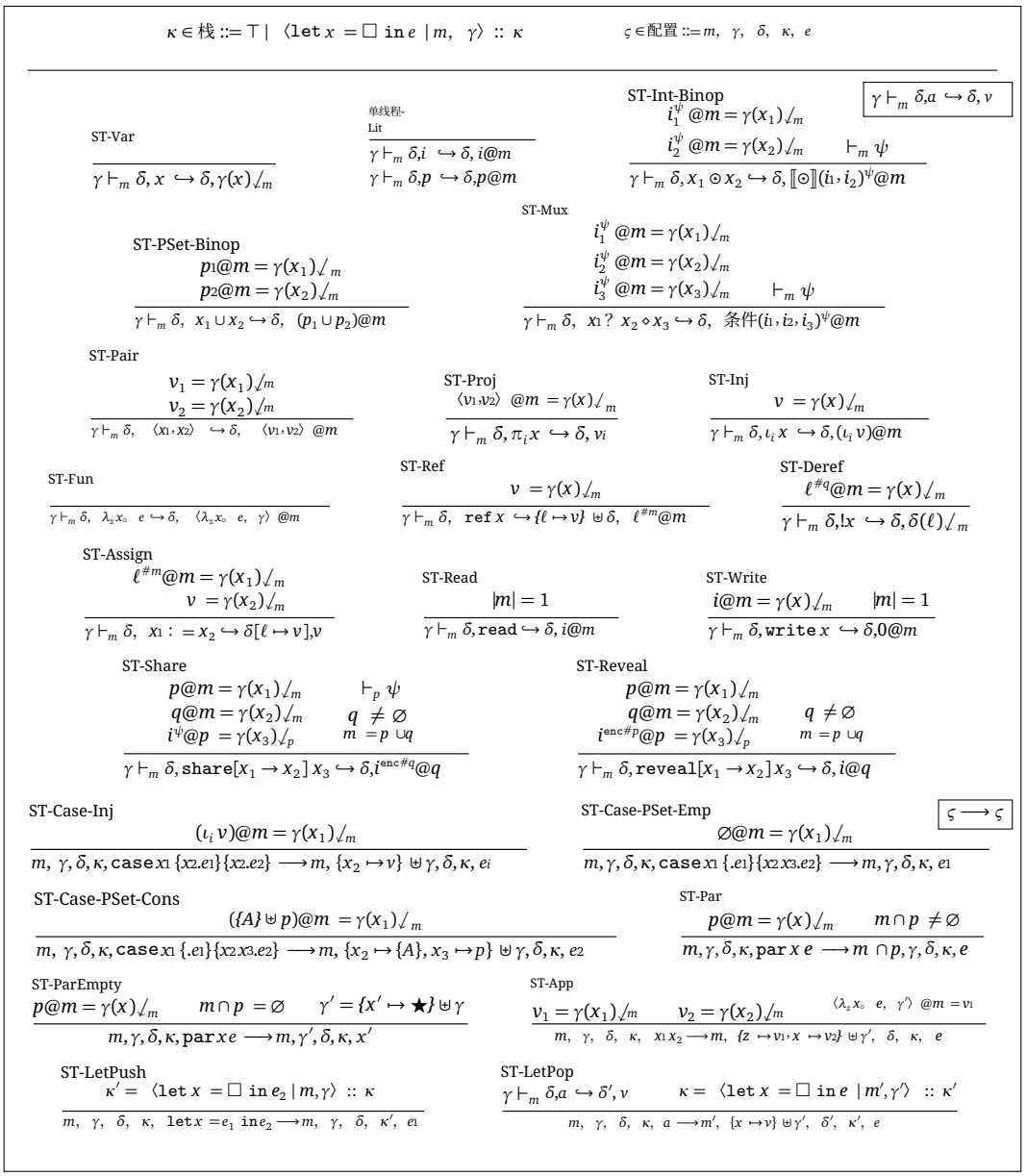
B.2 对应定理的证明概要

为了证明第 6 节给出的定理 6.1、定理 6.2 和定理 6.3，我们首先形式化关键定义。

定义 B.1 (终止状态)。

$$\begin{array}{lcl}
\varsigma \text{ is a terminal state} & \stackrel{\triangle}{\iff} & \varsigma = m, \gamma, \delta, \top, a \wedge \gamma \vdash_m \delta, a \hookrightarrow \delta', v \\
\dot{\varsigma} \text{ is a terminal state} & \stackrel{\triangle}{\iff} & \dot{\varsigma} = m, \dot{\gamma}, \dot{\delta}, \top, a \wedge \dot{\gamma} \vdash_m \dot{\delta}, a \hookrightarrow \dot{\delta}', \dot{v} \\
C \text{ is a terminal state} & \stackrel{\triangle}{\iff} & \forall A \in dom(C).\ C(A) \text{ is a terminal state}
\end{array}$$

此定义阐明了终止状态的含义：执行栈为空（$\top$），下一个要执行的项是原子的（a），并且原子表达式能够



■ **图 9** λ-Symphony 单线程语义。突出显示的前提 ■ 仅对惰性 MPC 求值是必需的。

通过 $\hookrightarrow$ 步进至一个值 v。从终止状态无法到达任何后继配置。我们将既非终止状态、又没有后继配置的状态称为卡住。

定义 B.2 (发散**)**。 若对于所有 ς'（其中 $\varsigma \longrightarrow^* \varsigma'$），都存在 ς'' 使得 $\varsigma' \longrightarrow \varsigma''$，则单线程配置 ς 是发散的。（分布式配置C 及其转换 $\rightsquigarrow^*$的情况也类似。）

定义 B.3 (局部卡住**)**。

C 局部卡住 $\stackrel{\Delta}{\Longleftrightarrow}$ ∃A，使得 $C(A) = \dot{\varsigma}$，且其中 $\dot{\varsigma}$ 不是终止状态$\dot{\varsigma}$。$e \notin \{share[\ \rightarrow\ ], reveal[\ \rightarrow\ ]\}\dot{\varsigma} \not\longrightarrow_A$ 或$\dot{\varsigma}$。
$e \in \{share[x_1 \rightarrow x_2]\, x3, reveal[x_1 \rightarrow x_2]\, x_3\}\, p = \dot{\varsigma}.\dot{\gamma}(x_1)$
$m = \dot{\varsigma}.m\ q = \dot{\varsigma}.\dot{\gamma}(x_2)\, m \neq p \cup q$

现在我们建立若干关键引理。我们关于定理 6.1 的证明方法基本遵循 Wysteria [28], , 而我们关于定理 6.2 和定理 6.3 的证明方法虽然是全新的，但通过对配置的递归语法以及归纳定义的关系 $\longrightarrow$、 $\rightsquigarrow$ 和 $\longrightarrow_A$进行分类讨论和归纳推理，可以直接证明。本节将展示总体的证明方法。

首先，我们建立单线程语义的确定性以及分布式语义的合流性：

引理 B.1 （单线程确定性）。 如果 $\varsigma \longrightarrow \varsigma_1$ 并且 $\varsigma \longrightarrow \varsigma_2$ 那么 $\varsigma_1 = \varsigma_2$。

证明。 对推导 $\varsigma \longrightarrow \varsigma_1$ 和 $\varsigma \longrightarrow \varsigma_2$进行分类讨论。 □

引理 B.2 （分布式合流性）。 如果 $C \rightsquigarrow^* C_1$ 并且 $C \rightsquigarrow^* C_2$，那么对于某个 C_3，有 $C_1 \rightsquigarrow^* C_3$ 并且$C_2 \rightsquigarrow^* C_3$ 。

证明。 我们首先证明一个菱形性质子引理，该引理表明：若$C \rightsquigarrow C_1$，$C \rightsquigarrow C_2$且$C_1 \neq C_2$，，则存在某个C_3，，使得$C_1 \rightsquigarrow C_3$ 且$C_2 \rightsquigarrow C_3$ 。对此，我们通过对推导$C \rightsquigarrow C_1$ 和 $C \rightsquigarrow C_2$进行分类讨论来证明。合流性是一个经典结果：满足菱形性质的迁移系统也是合流的。其证明对推导$C \rightsquigarrow^* C_1$ 和$C \rightsquigarrow^* C_2$ 进行归纳，并在基例中应用菱形性质。 □

接下来，我们证明终止状态与语义之间的前向模拟：

引理 B.3 （单线程正向模拟）。

1. 若 ς为终止的，则 $\varsigma\lightning$ 为终止的*2.* 若 ς卡住，则 $\varsigma\lightning$ 局部卡住*3.* 若 $\varsigma \longrightarrow^* \varsigma'$，则 $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$。

证明。

1. 对 ς进行分类讨论2. 对 ς进行分类讨论3. 对 $\varsigma \longrightarrow^* \varsigma'$ 中的步骤进行归纳，并对中间推导$\varsigma \longrightarrow \varsigma''$进行分类讨论。

□

定理 6.1 由以下引理直接推出：

ST/D 终止对应的证明。 正向表述等价于证明 $\varsigma \longrightarrow^* \varsigma'$ 和 ς' 终止蕴含 $\varsigma\lightning \longrightarrow^* \varsigma'\lightning$ 和 $\varsigma'\lightning$ 终止，这由引理 B.3 得出。

反向表述等价于证明 $\varsigma\lightning \rightsquigarrow^* C$ 和 C 终止蕴含对于某个 ς' 有 ς' 终止且 $C = \varsigma'\lightning$。根据引理 B.3 和引理 B.2，我们知道，如果 ς 发散，则 $\varsigma\lightning$ 必须发散；因此，在假设 $\varsigma\lightning$ 收敛的情况下，我们知道必须收敛，所以对于某个终止状态 ς' 有 $\varsigma \longrightarrow^* \varsigma'$。根据引理 B.3，我们知道 $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$；根据引理 B.2，我们知道 $C = \varsigma'\lightning$。

□

定理 6.2 的证明同样可由迄今为止证明的引理和定理推出：

ST/D 强非对称非终止对应的证明。

1. 根据定理 6.1，我们知道 ς 不会到达终止状态，因此它要么发散，要么收敛到卡住状态。下面分别考虑这两种情况。假设 ς 发散，则根据引理 B.3，我们知道存在一条同样发散的分布式轨迹。将引理 B.2 应用于卡住的分布式状态、刚刚确立的发散分布式状态以及作为共同祖先的 $\varsigma\lightning$ ，我们知道卡住的分布式状态可以向发散状态推进，这就产生了矛盾——因此该子情况不可能发生。另一种子情况是 ς 到达卡住状态，而这显然满足目标。

2. 由于根据引理 B.2， $\rightsquigarrow$ 具有合流性，因此 $\varsigma\lightning$ 必须收敛到终止状态、收敛到卡住状态或发散。（例如，不可能出现这样的情况： $\varsigma\lightning \rightsquigarrow^* \varsigma'$ 中 ς' 处于卡住状态，而 $\varsigma\lightning \rightsquigarrow^* \varsigma''$ 可以持续进行转换，却永远无法到达卡住状态或终止状态。）如果 $\varsigma\lightning$ 收敛，则根据定理 6.1 将导致矛盾。如果 $\varsigma\lightning$ 到达卡住状态，那么根据本定理的 (1)， ς 也会到达卡住状态，这将导致矛盾。因此， $\varsigma\lightning$必须发散。

□

在证明第三个定理之前，我们先证明最后一个引理：

引理 B.4 (D 局部停滞保持)。 若C 局部卡住，且$C \rightsquigarrow^* C'$ ，则C' 局部卡住。

证明。 对 $\rightsquigarrow^*$中的步数进行归纳，并对中间推导$C \rightsquigarrow C''$进行分类讨论。

□

因此，我们对定理 6.3 的证明使用了前面的引理：

卡住状态的 *ST/D* 正确性证明。我们假设 $\varsigma \longrightarrow^* \varsigma'$ 满足 ς' 卡住，且存在某个C 满足 $\varsigma\lightning \rightsquigarrow C$。我们必须证明存在$C'$，使得$C \rightsquigarrow C'$，且$C'$ 局部卡住。根据引理 B.3，我们知道 $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$，且 $\varsigma'\lightning$ 局部卡住。根据合流性，存在C'，使得$C \rightsquigarrow^* C'$，且$\varsigma'\lightning \rightsquigarrow^* C'$。根据引理 B.4，以 $\varsigma\lightning$ 为共同祖先，我们有C' 局部卡住。 □

定理 6.1 概括了先前工作（Wysteria [28]）所证明的相同元理论性质，而定理 6.2 和 6.3 则是对我们工作中首次提出的发散正确性和停滞状态正确性结果的细化。

B.3 关键引理的详细证明

本节证明分布式语义的关键元理论性质，即前向模拟（附录 B.3.1）和合流性（附录 B.3.2），以及它们的推论。完整的 DS 语义规则见图 10。

B.3.1 前向模拟

证明模拟的关键引理指出：如果全局单线程配置 ς 经过若干步转移到 ς'，那么对 ς 进行切片后，可通过多线程语义的若干步转移到达 ς'。证明的基本思路是根据全局配置 ς 的步转移形式，构造一系列分布式步转移，其中每一步都会更新模式 ς 下某个参与方的局部配置。对于非原子表达式，模式中的每个参与方恰好各有一步；最有趣的情况是全局步转移应用 SS-Par：这些步转移可由一系列步转移模拟，而这一系列步转移可以由 SS-Par 本身 或 SS-Empty 的应用构成。对于求值结果为原子项并绑定该结果的表达式，所有参与方共同执行一个步转移。

引理 B.5（前向模拟-步）。若 $\varsigma \to \varsigma'$，则 $\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$。

证明。 $\varsigma \to \varsigma'$，由假设可得。令 $(m,\ \gamma,\ \delta,\ \kappa,\ e) = \varsigma$，并令 $(m',\ \gamma',\ \delta',\ \kappa',\ e') = \varsigma'$。根据 $\varsigma \to \varsigma'$ 的证据形式进行分类讨论：

ST-Case-Inj $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$，其中每个 C_i 都是 $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$（其中$C|_I$ 表示将分布式配置 C 限制到索引为 I 的参与方）。

每个C_i 步进到C_{i+1} 的证明如下。应用 DS-Step，其中 $\dot{\varsigma}$ 作为配置

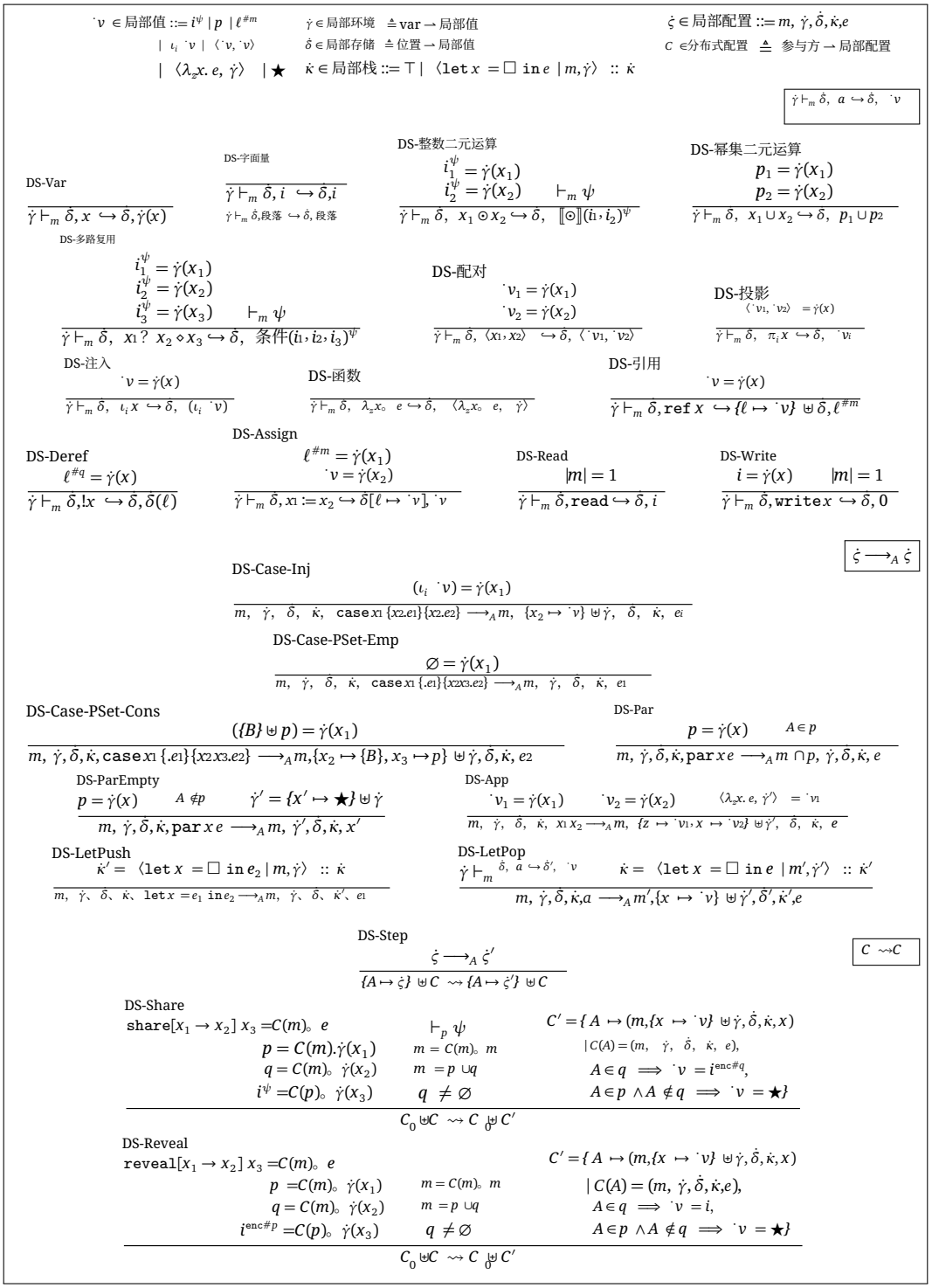
$$m_i, \gamma, \delta, \kappa, \text{case } x\{x_1.e_1\}\{x_2.e_2\}$$

$\dot{\varsigma}'$ 作为配置

$$m_i, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, e_j$$

其中 $\gamma(x) \swarrow_{m_i} = (\iota_j v)@\{m_i\}$ 以及 $C_i|_{[0,i-1],[i+1,|m|]}$作为$C_\circ$ $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$根据 DS-Case-Inj。$C_{i+1}$ 是$C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$。

由规则 DS-Case-PSet-Emp 和 DS-Case- PSet-Cons 构造的证据，其证明过程类似。唯一的区别是，每个分布式配置C_{i+1} 中更新后的局部配置，是通过更新表达式的主体而形成的



■ **图 10** λ-Symphony 分布式语义（完整图）。

在规则 DS-Case-PSet-Emp 的情况下为 e_1，在规则 DS-Case- PSet-Cons 的情况下为 e_2。此外，在规则 DS-Case-PSet-Cons 的情况下，局部状态会更新，将变量 x_2 和 x_3 绑定到解构后的主体和剩余参与方集合。

ST-Par $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$，其中每个 C_i 都是 $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$。每个 C_i 步进到 C_{i+1} 的证明如下。如果 $m_i \in p$，则应用 DS-Step，其中 $\dot{\varsigma}$ 作为配置

$$m_i, \gamma, \delta, \kappa, \texttt{par}\ p\ e$$

$\dot{\varsigma}'$ 作为配置

$$m_i, \gamma, \delta, \kappa, e$$

以及 $C_i|_{[0,i-1],\ [i+1,|m|]}$ 作为 $C_\circ$ $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ 根据 DS-Par，因为 $m_i \in p$，因此 $\{m_i\} \cap p = \{m_i\} \neq \emptyset_\circ$
如果 $m_i \notin p$，那么令 $\dot{\varsigma}' = \dot{\varsigma}_\circ$
在这两种情况下，C_{i+1} 是 $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]\circ}$

ST-ParEmpty $\varsigma\lightning \rightsquigarrow_\circ \ldots \rightsquigarrow C_i \rightsquigarrow_\circ \ldots \rightsquigarrow \varsigma'\lightning$，其中每个 C_i 是 $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]\circ}$ 证明每个 C_i 转移到 C_{i+1} 如下。应用 DS-Step，其中 $\dot{\varsigma}$ 作为配置

$$m_i, \gamma, \delta, \kappa, \texttt{par}\ p\ e$$

$\dot{\varsigma}'$ 作为配置

$$m_i, \{x \mapsto \bigstar\} \uplus \gamma, \delta, \kappa, x$$

且 $C_i|_{[0,i-1],[i+1,|m|]}$ 作为 $C_\circ$ $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ 根据 DS-ParEmpty，因为 $m \cap p = \emptyset$ 由以下事实可知：$c \rightarrow c$ 是 ST-ParEmpty 的一个应用；因此 $\{m_i\} \cap p = \emptyset_\circ$ C_{i+1} 是 $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]\circ}$

ST-App $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$，其中每个 C_i 都是 $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]\circ}$
每个 C_i 步进到 C_{i+1} 的证明如下。设 $\langle \lambda_z x.e', \gamma' \rangle\ @m = v_1 = \gamma(x_1)$，当 $c \rightarrow c'$ 是 ST-App 的一个应用时，这一结论成立。
应用 DS-Step，其中 $\dot{\varsigma}$ 作为配置

$$m_i, \gamma, \delta, \kappa, x_1\ x_2$$

$\dot{\varsigma}'$ 作为配置

$$m_i, \{z \mapsto v_1, x_2 \mapsto \gamma(x_2)\} \uplus \gamma, \delta, \langle \texttt{let}\ x\ =\ \_\ \texttt{in}\ e_2 \mid \gamma \rangle :: \kappa, e'$$

并且 $C_i|_{[0,i-1],[i+1,|m|]}$ 作为 $C_\circ$ $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ 依据 DS-App。C_{i+1} 是 $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]\circ}$

ST-LetPush $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$，其中每个 C_i 都是 $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]\circ}$
每个 C_i 步进至 C_{i+1} 的证明如下。应用 DS-Step，将 $\dot{\varsigma}$ 作为配置

$$m_i, \gamma, \delta, \kappa, \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2$$

$\dot{\varsigma}'$ 作为配置

$$m_i, \gamma, \delta, \langle \texttt{let } x = \_ \texttt{ in } e_2 \mid \rangle :: \kappa, e_1$$

并且$C_i|_{[0,i-1]}$, $[i+1,|m|]$ 作为$C_\circ$ $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ 由 DS-LetPush。C_{i+1} 是$C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$。

ST-LetPop e 是某个原子a，δ 并且a 执行一步到达 δ' 以及某个值v 在 γ 模式m下，并且 $\kappa = \langle$let 绑定$x = _$在$e' \mid m', \gamma'' \rangle :: \kappa'$，并且$\gamma' = \{x \mapsto v\} \uplus \gamma''$，根据假设。根据以下事实进行分类讨论：$\delta$ 与a 在模式m下单步规约为 δ'，并得到某个值v γ。**子情况：单独原子**若该求值应用了 ST-Int、ST-Var、ST-Fun、ST-Inj、ST-Pair、ST-Proj、ST-Ref、ST-Deref、ST-Assign、ST-Fold、ST-Unfold、ST-Read、ST-Write、ST-Embed 和 ST-Star 中的规则，则有 $\varsigma\lightning \rightsquigarrow \ldots \rightsquigarrow C_i \rightsquigarrow \ldots \rightsquigarrow \varsigma'\lightning$，其中每个$C_i$ 都是 $\varsigma\lightning|_{[0,i]} \uplus \varsigma'\lightning|_{[i+1,|m|]}$。证明每个$C_i$ 单步规约为C_{i+1} 如下。应用 DS-Step，其中 $\dot{\varsigma}$作为配置

$$m_i, \gamma, \delta, \langle \texttt{let } x = \_ \texttt{ in } e' \mid m', \gamma'' \rangle :: \kappa', a$$

$\dot{\varsigma}'$ 作为配置

$$m_i, \{x \mapsto v\} \uplus \gamma, \delta', \kappa', e'$$

且 $C_i|_{[0,i-1]}$, $[i+1,|m|]$ 作为 $C_\circ$ $\dot{\varsigma} \longrightarrow_i \dot{\varsigma}'$ 由 DS-LetPop 给出。C_{i+1} 是 $C_i|_{[0,i-1]} \uplus \{m_i \mapsto \dot{\varsigma}'\} \uplus C_i|_{[i+1,|m|]}$。

子情况：明文数据上的二元运算 求值步骤应用 ST-Binop 的子情况中，a 的形式为 $x_1 \oplus x_2, i_1 @ m = \gamma(x_1) \swarrow_m$，且$i_2 @ m = \gamma(x_1) \swarrow_m$，或者求值步骤应用 ST-PSet-Binop（即计算是明文数据上的二元运算）的子情况，与前一个子情况直接类似。

子情况：对明文数据进行多路复用求值是 ST-Mux 应用的子情况 a 的形式为 如果 x $\gamma(\_2) \swarrow_m$ $= \gamma(\_3) \swarrow_m$ 那么 x否则 $x i_1$ $@m = \gamma(x_1) \swarrow_m$，$i_2 @ m\, x$，并且 $i @ m\, x$与前面的子情况直接类似。

子情况：对加密数据进行二元运算 对于求值是 ST-Binop 应用的子情况，a 的形式为 $x_1 \oplus x_2, i_{\text{enc\#}m\,1} @ m = \gamma(x_1) \swarrow_m$，并且$i_{\text{enc\#}m\,2} @ m = \gamma(x_1) \swarrow_m$（即，计算是在加密数据上进行二元运算），$\varsigma\lightning$ 通过应用 DS-Step 将步骤执行至 $\varsigma'\lightning$，其中

$$m, \gamma, \delta, \kappa, x_1 \oplus x_2$$

作为 ς,

$$m, \{x \mapsto v\} \uplus \gamma, \delta, \kappa, x$$

作为 $\dot{\varsigma}'$，且 $\varsigma\lightning|_{\text{parties}\setminus m}$ 作为 $C_\circ$。$\dot{\varsigma}$通过 ST-Binop 执行步骤到 $\dot{\varsigma}'$。

子情况：对加密数据进行多路复用。当求值应用 ST-Mux 时，a 的形式为 mux if x_1 then x_2 else x_3, ienc#m 1 @m $= \gamma(x_1) \swarrow_m$, ienc#m 2 @m $= \gamma(x_2) \swarrow_m$, and ienc#m 3 @m $= \gamma(x_3) \swarrow_m$，该子情况与前一个子情况直接类似。

子情况：同步 求值应用 ST-Share 或 ST-Reveal 的子情况与前两个子情况直接类似，因为它们由分布式语义中的一步模拟。

□

弱向前模拟的证明可直接由引理 B.5 得出。

引理 B.6 (单线程弱向前模拟)。 如果 $\varsigma \longrightarrow^* \varsigma'$ 和 ς' 是终止的，那么$\varsigma\lightning \rightsquigarrow^* \varsigma'\lightning$ 和 $\varsigma'\lightning \not\rightsquigarrow$。

证明。 根据多步判断 $\varsigma \longrightarrow^* \varsigma$，对其应用归纳法即可证明该命题。
空 如果轨迹为空，则 ς是 ς'。 $\varsigma\lightning$ 通过空步骤序列多步到达 $\varsigma\lightning$ 。

非空 如果轨迹的形式为 $\varsigma \rightarrow \varsigma'' \rightarrow^* \varsigma'$，那么根据引理 B.5 可得 $\varsigma\lightning \rightsquigarrow^* \varsigma''$，根据归纳假设可得$\varsigma''\lightning \rightsquigarrow^* \varsigma'\lightning$ 。 根据两条轨迹的串接仍是一条轨迹这一事实可得 $\varsigma\lightning \rightsquigarrow^* \varsigma\lightning$ 。

□

B.3.2 合流性与终态确定性

为了证明菱形性质，我们将首先提出并证明一个引理，该引理表明：在分布式配置的每一步中，能够执行一步的不同子配置，实际上会更新属于互不相交的参与方集合的局部配置。

引理 B.7. 对于所有分布式配置 C、C_0，和 C_1 ， 以及所有非终止分布式配置 C_0' 和 C_1' ， 若满足

$$C = C_0' \uplus C_0 = C_1' \uplus C_1$$

以下情况之一成立：

1. $C_0' = C_1'$ 且 $C_0 = C_1$；2. C_0' 和 C_1' 的定义域互不相交。

证明。 根据 C_0' 和 C_1' 的定义域是否互不相交进行分情况讨论。若是，则满足该断言的第二个条件。

否则，存在某个参与方 m_i ，同时属于 C_0' 和 C_1' 的定义域。根据局部配置中位于 m_i 处的活动表达式 e_i 分类讨论，C_0' 和 C_1' 的定义域相同：若 e_i 是非原子项、变量出现、整数字面量、整数二元运算、主体集合二元运算、多路复用、二元组创建、二元组投影、和类型注入、函数创建、引用创建、解引用、引用赋值、递归类型引入、读取或写入，则定义域都是单例集合。因此，由于这两个定义域都是相交的单例集合，它们相同。

若表达式将一个值从 p 共享到 q，则从 C_0' 和 C_1' 出发的步骤都是 DS-Share 的应用，而 DS-Share 的一个前提是模式为 $p \cup q$；因此，C_0' 和 C_1' 的定义域是相同的参与方集合 $p \cup q$。

在表达式向参与方 q揭示绑定到变量 x 的值时，从 C_0' 和 C_1' 出发的步骤是 DS-Reveal 的应用，其前提是绑定到 x 的值已加密给参与方 p，且活跃参与方为 $p \cup q$；因此，C_0' 和 C_1' 的定义域是相同的参与方集合 $p \cup q$。因此，C_0 和 C_1 是

相同的，因为它们分别是将 C 限制到 C_0' 和 C_1' 的定义域的补集所得。

□

利用引理 B.7，我们可以证明多线程配置上的转换关系具有菱形性质。

证明。存在分布式配置 $C_{0,0}$，$C_{0,1}$，$C_{1,0}$，和 $C_{1,1}$，使得

$$C_{0,0} \uplus C_{0,1} = C = C_{1,0} \uplus C_{1,1}$$

and

$$C_0 = C_{0,0} \uplus C_{0,1}'$$

$$C_1 = C_{1,0} \uplus C_{1,1}'$$

根据 $C_{0,1} \rightsquigarrow C_{0,1}'$ 和 $C_{1,1} \rightsquigarrow C_{1,1}'$，通过反演 C 转换到 C_0 以及 C 转换到 C_1这两个事实即可得到。根据引理 B.7 应用于 C、$C_{0,0}$，$C_{0,1}$，$C_{1,0}$，和 $C_{1,1}$的情况进行分类：**相同** 立即可得 $C_{0,0} = C_{1,0}$。此外，直接分析多线程转换关系可得 $C_{0,1}' = C_{1,1}'$。因此

$$C_0 = C_{0,0} \uplus C_{0,1}' = C_{1,0} \uplus C_{1,1}' = C_1$$

由同余可得。因此，对于$C' = C_0' = C_1'$，$C \rightsquigarrow C_0'$ 和$C \rightsquigarrow C_1'$ 均成立。

互不相交。令C'' 为C 在$C_{0,0}$ 中的参与方和$C_{1,0}$ 上的限制，并令

$$C' = C'' \uplus C_{0,1}' \uplus C_{1,1}'$$

C' 定义良好，因为$C_{0,1}'$ 和$C_{1,1}'$的定义域分别是$C_{0,1}$和$C_{1,1}$，的定义域；根据本子句的假设，它们互不相交。

根据$C_0 \rightsquigarrow C_0'$这一事实，按情况讨论$C_0 \rightsquigarrow C'$：在每种情况下，调整证明，使其使用$C'' \uplus C_1$ 作为保持不变并与C_0连接的分布式配置。$C_1 \rightsquigarrow C'$ 可由对称论证得出。

□

鉴于分布式语义满足菱形性质，合流性（引理 B.8）是一般转换系统和重写系统基本性质的直接推论。

引理 B.8（DS 多步合流性）。如果$C \rightsquigarrow^* C_1$ 且$C \rightsquigarrow^* C_2$，那么存在C_3，使得$C_1 \rightsquigarrow^* C_3$ 且$C_2 \rightsquigarrow^* C_3$。

证明。将满足菱形性质的任何二元关系都满足合流性这一事实 [3] 应用于分布式步关系的菱形性质。□

合流性的一个直接推论是，从同一状态到达的所有停机状态都相同。

推论 B.8.1（DS 终态确定性）。如果 $C \rightsquigarrow^* C_1$ 且 $C \rightsquigarrow^* C_2$，$C_1 \not\rightsquigarrow$ 且$C_2 \not\rightsquigarrow$ 那么 $C_1 = C_2$。

证明。存在某个分布式配置 C'，使得 $C_1 \rightsquigarrow^* C'$ 且 $C_2 \rightsquigarrow^* C'$；这是将引理 B.8 应用于以下事实得出的：$C \rightsquigarrow^* C_1$ 且 $C \rightsquigarrow^* C_2$。根据事实 C_1 会停机，因此 C_1 经由空步骤序列多步转移至 C'，所以 C' 是 C_1；通过对称论证可知，C'是 C_2。因此，C_1 是 C_2。

□

引用

[1] Coşku Acay、Rolph Recto、Joshua Gancher、Andrew C. Myers 和 Elaine Shi。Viaduct：一种用于安全分布式程序的可扩展优化编译器。载于《编程语言设计与实现》，PLDI 2021。计算机协会，2021年。doi:10.1145/3453483.3454074。[2] Abdelrahaman Aly、Marcel Keller、Dragos Rotaru、Peter Scholl、Nigel P. Smart 和 Tim Wood。SCALE-MAMBA，2019年。访问日期：2022年1月22日。网址：https://homes.esat.kuleuven.be/~nsmart/SCALE/。[3] Franz Baader 和 Tobias Nipkow。《项重写及其他》。剑桥大学出版社，1999年。doi:10.1017/CBO9781139172752。[4] Michael Ben-Or、Shafi Goldwasser 和 Avi Wigderson。非密码学容错分布式计算的完备性定理。载于《计算理论研讨会》，STOC 1988。计算机协会，1988年。doi:10.1145/62212.62213。[5] Lennart Braun、Daniel Demmler、Thomas Schneider 和 Oleksandr Tkachenko。MOTION——混合协议多方计算框架。《ACM隐私与安全汇刊》，25(2)，2022年。doi:10.1145/3490390。[6] Paul Bunn、Jonathan Katz、Eyal Kushilevitz 和 Rafail Ostrovsky。高效的三方分布式 ORAM。载于《网络安全与密码学》，SCN 2020。Springer International Publishing，2020年。doi:10.1007/978-3-030-57990-6_11。[7] Niklas Büscher、Daniel Demmler、Stefan Katzenbeisser、David Kretzmer 和 Thomas Schneider。HyCC：面向实用安全计算的混合协议编译。载于《计算机与通信安全会议》，CCS 2018。计算机协会，2018年。doi:10.1145/3243734.3243786。[8] Luís Cruz-Filipe 和 Fabrizio Montesi。编舞式编程的核心模型。载于《组件软件形式化方面》，FACS 2016。Springer International Publishing，2016年。doi:10.1007/978-3-319-57666-4_3。[9] Luís Cruz-Filipe 和 Fabrizio Montesi。过程式编舞编程。载于《分布式对象、组件与系统的形式化技术》，FORTE 2017。Springer International Publishing，2017年。doi:10.1007/978-3-319-60225-7_7。[10] Mila Dalla Preda、Maurizio Gabbrielli、Saverio Giallorenzo、Ivan Lanese 和 Jacopo Mauro。动态编舞。载于《协调模型与语言》，COORDINATION 2015。Springer International Publishing，2015年。doi:10.1007/978-3-319-19282-6_5。[11] Ivan Damgård、Valerio Pastro、Nigel P. Smart 和 Sarah Zakarias。基于部分同态加密的多方计算。载于《密码学进展》，CRYPTO 2012。Springer International Publishing，2012年。doi:10.1007/978-3-642-32009-5_38。[12] Daniel Demmler、Thomas Schneider 和 Michael Zohner。ABY——高效混合协议双方计算框架。载于《网络与分布式系统安全研讨会》，NDSS 2015。互联网协会，2015年。doi:10.14722/ndss.2015.23113。

[13] Brett Hemenway Falk 和 Rafail Ostrovsky。使用 o(n log log n) 次安全操作进行安全归并。载于《信息论密码学》，ITC 2021。Schloss Dagstuhl – Leibniz-Zentrum für Informatik，2021。doi:10.4230/LIPIcs.ITC.2021.7。[14] Martin Franz、Andreas Holzer、Stefan Katzenbeisser、Christian Schallhart 和 Helmut Veith。CBMC-GC：用于安全两方计算的 ANSI C 编译器。载于《编译器构造》，CC 2014。Springer International Publishing，2014。doi: 10.1007/978-3-642-54807-9_15。[15] Oded Goldreich。迈向软件保护与通过不经意随机存取存储器进行模拟的理论。载于《计算理论研讨会》，STOC 1987。计算机协会，1987。doi:10.1145/28395.28416。[16] Oded Goldreich、Silvio Micali 和 Avi Wigderson。如何进行任意心理博弈，或：诚实多数协议的完备性定理。载于《计算理论研讨会》，STOC 1987。计算机协会，1987。doi:10.1145/28395.28420。[17] Koki Hamada、Ryo Kikuchi、Dai Ikarashi、Koji Chida 和 Katsumi Takahashi。基于比较排序算法的高效实用多方排序协议。载于《信息安全与密码学》，ICISC 2012。Springer International Publishing，2012。doi:10.1007/978-3-642-37682-5_15。[18] Andrew K. Hirsch 和 Deepak Garg。Pirouette：高阶类型化函数编舞。载于《编程语言原理》，POPL 2022。计算机协会，2022。doi:10.1145/3498684。[19] Martin Hirt 和 Ueli Maurer。完美多方计算中的参与方模拟与一般对手结构。《密码学杂志》，13(1)，2000。doi: 10.1007/s001459910003。[20] Yan Huang、David Evans 和 Jonathan Katz。私有集合求交：混淆电路是否优于定制协议？载于《网络与分布式系统安全研讨会》，NDSS 2012。互联网协会，2012。[21] Muhammad Ishaq、Ana L. Milanova 和 Vassilis Zikas。通过程序分析实现高效多方计算：高效最优混合框架。载于《计算机与通信安全》，CCS 2019。计算机协会，2019。doi:10.1145/3319535.3339818。[22] Florian Kerschbaum。自动优化安全计算。载于《计算机与通信安全》，CCS 2011。计算机协会，2011。doi:10.1145/2046707.2046786。[23] Sven Laur、Jan Willemson 和 Bingsheng Zhang。轮次高效的不经意数据库操作。载于《信息安全》，ISC 2011。Springer International Publishing，2011。doi:10.1007/978-3-642-24861-0_18。[24] Chang Liu、Xiao Shaun Wang、Kartik Nayak、Yan Huang 和 Elaine Shi。ObliVM：用于安全计算的编程框架。载于《安全与隐私研讨会》，SP 2015。IEEE 计算机学会出版社，2015。doi:10.1109/SP.2015.29。[25] Fabrizio Montesi。编舞式编程。博士论文，2013。[26] Benjamin Mood、Debayan Gupta、Henry Carter、Kevin Butler 和 Patrick Traynor。Frigate：一种经过验证、可扩展且高效的安全计算编译器与解释器

。载于欧洲安全与隐私研讨会，EuroSP 2016。IEEE 计算机学会出版社，2016。doi:10.1109/EuroSP.2016.20。[27] Jaak Randmets。安全多方计算应用开发编程语言。博士论文，2017。[28] Aseem Rastogi、Matthew A. Hammer 和 Michael Hicks。Wysteria：一种用于通用混合模式多方计算的编程语言。载于安全与隐私研讨会，SP 2014。IEEE 计算机学会出版社，2014。doi:10.1109/SP.2014.48。[29] Aseem Rastogi、Piotr Mardziel、Michael Hicks 和 Matthew A. Hammer。通过知识推断优化安全多方计算。载于安全编程语言与分析，PLAS 2013。计算机协会，2013。doi:10.1145/2465106.2465117。[30] Aseem Rastogi、Nikhil Swamy 和 Michael Hicks。Wys*：一种用于经验证的安全多方计算的领域特定语言。载于安全与信任原理，POST 2019。Springer International Publishing，2019。doi:10.1007/978-3-030-17138-4_5。[31] Andrei Sabelfeld 和 Andrew C Myers。基于语言的信息流安全。*IEEE*通信领域精选期刊，21(1)，2003。doi:10.1109/JSAC.2002.806121。[32] Berry Schoenmakers。MPyC：Python 中的安全多方计算，2019。访问日期：2022年1月22日。URL：https://github.com/lschoe/mpyc。[33] Ebrahim M. Songhori、Siam U. Hussain、Ahmad-Reza Sadeghi、Thomas Schneider 和 Farinaz Koushanfar。TinyGarble：高度压缩且可扩展的顺序混淆电路。载于安全与隐私研讨会，SP 2015。IEEE 计算机学会出版社，2015。doi:10.1109/SP.2015.32。[34] Abraham Waksman。一种置换网络。《*ACM*学报》，15(1)，1968。doi:10.1145/321439.321449。[35] Xiao Wang、Hubert Chan 和 Elaine Shi。Circuit ORAM：论 Goldreich-Ostrovsky 下界的紧致性。载于计算机与通信安全，CCS 2015。计算机协会，2015。doi:10.1145/2810103.2813634。[36] Xiao Wang、Alex J. Malozemoff 和 Jonathan Katz。EMP-toolkit：高效多方计算工具包，2016。访问日期：2022年1月22日。URL：https://github.com/emp-toolkit。[37] Andrew Chi-Chih Yao。如何生成并交换秘密。载于计算机科学基础研讨会，SFCS 1986。IEEE 计算机学会出版社，1986。doi:10.1109/SFCS.1986.25。[38] Qianchuan Ye 和 Benjamin Delaware。不经意代数数据类型。载于编程语言原理，POPL 2022。计算机协会，2022。doi:10.1145/3498713。[39] Samee Zahur 和 David Evans。Obliv-C：一种支持扩展的数据无关计算语言。密码学电子印本档案，论文 2015/1153，2015。访问日期：2022- 01-22。URL：https://eprint.iacr.org/2015/1153。

[40] Yihua Zhang、Aaron Steele 和 Marina Blanton。《PICCO：用于隐私分布式计算的通用编译器》。发表于计算机与通信安全会议，CCS 2013。计算机协会，2013 年。doi:10.1145/2508859.2516752。

关于作者

Ian Sweetisweet@galois.com

David Daraisdarais@galois.com

David Heathheath.davidanthony@gatech.edu

William Harriswharris@galois.com

Ryan Estesrestes@uvm.edu

Michael Hicks Michael Hicks（mwh@cs.umd.edu）现就职于亚马逊。此项工作是在马里兰大学帕克分校任职期间完成的。