(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 104335218 A
(43)申请公布日 2015.02.04

(21)申请号 201380028121.0

(22)申请日 2013.03.28

(30)优先权数据
61/617,991 2012.03.30 US
61/618,010 2012.03.30 US

(85)PCT国际申请进入国家阶段日
2014.11.28

(86)PCT国际申请的申请数据
PCT/CA2013/000305 2013.03.28

(87)PCT国际申请的公布数据
WO2013/142981 EN 2013.10.03

(71)申请人 爱迪德加拿大公司
地址 加拿大安大略省

(72)发明人 H.约翰逊 Y.X.古 M.韦纳 Y.周

(74)专利代理机构 中国专利代理(香港)有限公司 72001
代理人 张凌苗 陈岚

(51)Int.Cl.
G06F 21/52(2006.01)
G06F 21/14(2006.01)
G06F 21/54(2006.01)
G06F 9/44(2006.01)

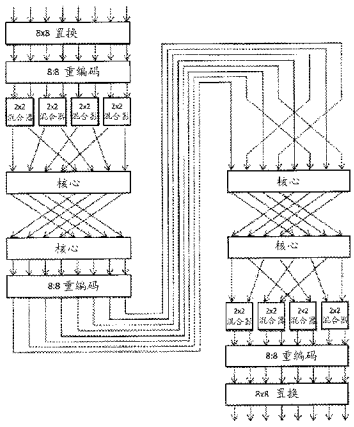
权利要求书5页 说明书112页 附图27页

(54)发明名称

使用基函数编码来保护可访问的系统

(57)摘要

提供了用于保护可访问的计算机可执行程序代码和系统的系统与技术。一个或多个基函数可以被生成并与现有程序代码掺合，使得潜在攻击者可能难以或者不可能从现有代码中区分基函数。还可以使用各种其他掺合和保护技术来保护所述系统和代码，所述其他掺合和保护技术诸如分式、变量相关编码、动态数据识别编码以及交叉链接，它们可以单独或组合使用，和/或可以与基函数掺合。



CN 104335218 A

1. 一种方法，包括：

选择字大小 w；

选择矢量长度 N；

生成可逆状态矢量函数，所述可逆状态矢量函数被配置为对 w 元素字的 N 矢量进行操作，所述可逆状态矢量函数包括多个可逆操作的组合，其中所述状态矢量函数接收至少 64 比特的输入并提供至少 64 比特的输出，并且所述状态矢量函数中的步骤的第一部分在 $Z/(2^w)$ 上执行线性或仿射计算；

使用第一编索引技术对所述状态矢量函数中的步骤的第一部分编索引；

使用第二编索引技术对所述状态矢量函数中的步骤的第二部分编索引；

在现有计算机可执行程序中选择至少一个操作以进行修改；以及

修改所述现有计算机程序以取代所选择的至少一个操作而执行所述状态矢量函数。

2. 如权利要求 1 所述的方法，其中，所述第一编索引技术和所述第二编索引技术中的每一个控制从由以下构成的组独立选择的操作类型：if-then-else 构造；switch 构造；元素置换选择；迭代计数；元素旋转计数；以及函数索引的密钥索引。

3. 如权利要求 1 或 2 中任意一项所述的方法，其中，所述状态矢量函数中的步骤的第三部分中的每个步骤包括非 T 函数操作。

4. 如权利要求 3 所述的方法，其中步骤的所述第三部分中的每个步骤是从由以下构成的组中选择的操作类型：函数索引的加密钥的按元素旋转以及函数索引的加密钥的子矢量置换。

5. 如任何之前权利要求所述的方法，其中，所述可逆状态矢量函数包括多个可逆操作的串接。

6. 如任何之前权利要求所述的方法，其中，w 是从由以下构成的组中选择的：16 比特、32 比特以及 64 比特。

7. 如任何之前权利要求所述的方法，其中，w 被选择为主计算平台的默认整数大小。

8. 如任何之前权利要求所述的方法，其中，字大小 w 是 N 矢量的内部字大小的两倍。

9. 如任何之前权利要求所述的方法，还包括：

生成所述可逆状态示例函数的逆，所述可逆状态矢量函数的逆包括所述多个可逆操作的每个的逆的串接。

10. 如任何之前权利要求所述的方法，还包括：

从由以下构成的组中选择用于所述可逆状态矢量函数的密钥类型：运行时密钥、生成时密钥以及函数索引的密钥。

11. 如权利要求 10 所述的方法，其中，所选择的密钥类型是运行时密钥，所述方法还包括：

修改所述状态矢量函数以接受提供密钥 k 的运行时输入。

12. 如权利要求 10 所述的方法，其中，所选择的密钥类型是生成时密钥，所述方法还包括针对密钥 K 部分地评估所述状态矢量函数。

13. 如权利要求 10 所述的方法，其中，所选择的密钥类型是函数索引的密钥，所述方法还包括：针对所述多个可逆操作 A 的每一个，提供用于可逆操作的相关联逆的密钥 R_A。

14. 如任何之前权利要求所述的方法，其中，所述状态矢量函数是至少部分通过多个矩阵操作来实现的。

15. 如任何之前权利要求所述的方法，其中，所述第一编索引技术和所述第二编索引技术中的至少一个控制多个操作，所述多个操作包括根据排序网络拓扑执行的随机交换。

16. 如权利要求 15 所述的方法，其中，所述排序网络拓扑是从由以下构成的组中选择的：Batcher 网络、Banyan 网络、完美混洗网络以及 Omega 网络。

17. 如任何之前权利要求所述的方法，还包括：

用第一编码机制对至所述状态矢量函数的输入进行编码；

其中，所述状态矢量函数中的每个操作被适配和配置为当至所述状态矢量函数的所述输入用不同于所述第一编码机制的第二编码机制编码时操作。

18. 如权利要求 17 所述的方法，其中，所述第一编码机制将所述输入编码为 aM+b，其中 a 和 b 是常数。

19. 如权利要求 18 所述的方法，其中，M 是可逆矩阵。

20. 如权利要求 18 或 19 中任意一项所述的方法，其中，所述第二编码机制当应用于所述输入时将所述输入编码为 cN+d，其中 c 和 d 分别是不同于 a 和 b 的常数。

21. 如权利要求 20 所述的方法，其中，N 是可逆矩阵。

22. 如任何之前权利要求所述的方法，其中，在所述现有计算机可执行程序中的所述至少一个操作和所述状态矢量函数使用计算上类似的运算。

23. 如任何之前权利要求所述的方法，其中，修改所述现有计算机程序的步骤还包括向所述状态矢量函数和所述现有计算机程序的组合应用从由以下构成的组中选择的至少一种技术：分式、变量相关编码、动态数据识别编码以及交叉链接。

24. 如权利要求 23 所述的方法，其中，所述状态矢量函数和实现所述至少一种技术代码中的每一个使用在计算上与所述现有计算机程序中给出的那些类似的运算。

25. 一种方法，包括：

接收具有字大小 w 的输入；

向所述输入应用可逆状态矢量函数，所述可逆状态矢量函数被配置为对 w 元素字的 N 矢量进行操作，所述可逆状态矢量函数包括多个可逆操作的组合，其中所述状态矢量函数中的步骤的第一部分在 $Z/(2^w)$ 上执行线性或仿射计算；

向所述可逆状态矢量函数的输出应用来自第一多个操作当中的第一操作，所述第一操作是基于第一编索引技术而选择的；

向所述第一操作的输出应用来自第二多个操作当中的第二操作，所述第二操作是基于与所述第一编索引技术不同的第二编索引技术而选择的；以及

提供所述第二操作的输出。

26. 如权利要求 25 所述的方法，其中，所述第二操作是基于从所述第一操作的执行得到的索引而从所述第二多个操作选择的。

27. 如权利要求 25 或 26 中任意一项所述的方法，其中，所述第一编索引技术和所述第二编索引技术中的每一个控制从由以下构成的组独立选择的操作类型：if-then-else 构造；switch 构造；元素置换选择；迭代计数；元素旋转计数；以及函数索引的密钥索引。

28. 如权利要求 25-27 中任意一项所述的方法，其中，所述状态矢量函数中的步骤的第

三部分中的每个步骤包括非 T 函数操作。

29. 如权利要求 28 所述的方法，其中步骤的所述第三部分中的每个步骤是从由以下构成的组中选择的操作类型：函数索引的加密钥的按元素旋转以及函数索引的加密钥的子矢量置换。

30. 如权利要求 25-29 中任意一项所述的方法，其中，所述可逆状态矢量函数包括多个可逆操作的串接。

31. 如权利要求 25-30 中任意一项所述的方法，其中，w 是从由以下构成的组中选择的：16 比特、32 比特以及 64 比特。

32. 一种通过执行第二操作来执行第一操作的方法，所述方法包括：

通过以下来执行所述第二操作：

接收用第一编码 A 编码为 A(X) 的输入 X；

使用$B^{-1}(X)$的值对所述输入执行第一多个计算机可执行操作，其中B^{-1}是第二编码机制 B 的逆，所述第二编码 B 不同于所述第一编码 A；以及

基于$B^{-1}(X)$来提供输出。

33. 如权利要求 32 所述的方法，其中，所述第一操作是对$B^{-1}\circ A(X)$执行的。

34. 如权利要求 33 所述的方法，其中，所述第一操作的输出未被在外部提供给可执行代码，所述第一操作与所述可执行代码被集成在一起。

35. 如权利要求 32-34 中任意一项所述的方法，其中，所述第一编码机制将所述输入编码为 aM+b，其中 a 和 b 是常数。

36. 如权利要求 35 所述的方法，其中，M 是可逆矩阵。

37. 如权利要求 35 或 36 中任意一项所述的方法，其中，所述第二编码机制如果应用于所述输入则将所述输入编码为 cN+d，其中 c 和 d 分别是不同于 a 和 b 的常数。

38. 如权利要求 37 所述的方法，其中，N 是可逆矩阵。

39. 一种方法，包括：

对于被配置为接收输入并提供输出的矩阵操作，在执行操作之前，根据排序网络拓扑来置换所述输入；

使用置换后的输入执行所述矩阵操作以生成输出；

根据所述排序网络拓扑来置换所述输出；以及

将置换后的输出提供为所述矩阵操作的输出。

40. 如权利要求 39 所述的方法，其中，所述排序网络拓扑是从由以下构成的组中选择的：Batcher 网络、Banyan 网络、完美混洗网络以及 Omega 网络。

41. 如权利要求 39 或 40 中任意一项所述的方法，其中，对于多个后续操作中的每一个，根据所述排序网络拓扑来置换针对后续操作的输入。

42. 一种方法，包括：

接收第一输入；

向所述第一输入应用函数索引的交织的第一函数以生成具有左部分和右部分的第一输出；

向所述第一输出应用函数索引的交织的第二函数以生成第二输出，其中，所述第一输

出的左部分被用作所述第二函数的右输入，并且所述第一输出的右部分被用作所述第二函数的左输入；以及

提供所述第二输出作为对所述第一输入的编码。

43. 如权利要求 42 所述的方法，其中，所述第一输入用第一编码来编码，还包括：

基于不同于所述第一编码的第二编码来应用函数索引的交织的第一函数和函数索引的交织的第二函数。

44. 如权利要求 42 所述的方法，还包括：

用第一编码对输入编码；以及

使用被适配和配置为当所述第一输入用不同于第一编码机制的第二编码机制来编码时对输入进行操作的操作，来执行每个其他记载的步骤。

45. 如权利要求 44 所述的方法，其中，所述第一编码机制将所述第一输入编码为 aM+b，其中 a 和 b 是常数。

46. 如权利要求 45 所述的方法，其中，M 是可逆矩阵。

47. 如权利要求 45 所述的方法，其中，所述第二编码机制如果应用于所述第一输入则将输入编码为 cN+d，其中 c 和 d 分别是不同于 a 和 b 的常数。

48. 如权利要求 47 所述的方法，其中，N 是可逆矩阵。

49. 一种方法，包括：

生成密钥 K；

基于所生成的密钥 K 和随机化信息 R 来生成一对基函数 f_K f_K^{-1}；

向通信管道的第一端应用基函数 f_K；

向所述通信管道的第二端应用基函数逆 f_K^{-1}；以及

丢弃所述密钥 K。

50. 如权利要求 50 所述的方法，其中，所述密钥 K 是使用随机或伪随机过程生成的。

51. 如权利要求 49-50 中任意一项所述的方法，其中，所述通信管道的第一端通过在第一平台上的第一应用来访问。

52. 如权利要求 49-51 中任意一项所述的方法，其中，所述通信管道的第二端通过在所述第一平台上的第二应用来访问。

53. 如权利要求 49-51 中任意一项所述的方法，其中，所述通信管道的第二端通过在第二平台上的第二应用来访问。

54. 一种方法，包括：

接收至少一个基函数；

接收用于现有计算机程序的应用代码；以及

通过用所述至少一个基函数替换所述应用代码中的至少一个操作，来掺合所述至少一个基函数和用于所述现有计算机程序的所述应用代码。

55. 如权利要求 54 所述的方法，还包括：

向所述至少一个基函数和所述应用代码应用至少一种掺合技术，所述至少一种掺合技术选自由以下构成的组：分式、变量相关编码、动态数据识别编码以及交叉链接。

56. 一种计算机系统，包括：

处理器；以及

存储指令的计算机可读存储介质，所述指令使得所述处理器执行任何之前权利要求中所记载的方法。

使用基函数编码来保护可访问的系统

技术领域

[0001] 本发明总地涉及电子计算设备和计算机系统，并且更特别地，涉及保护是可访问的而受攻击的设备和系统上的软件和固件。

背景技术

[0002] 计算机、电子计算设备以及计算机软件在所有它们各种形式上的使用被认为是非常普遍的并且日益增长。此外，随着强大通信网络的普遍，计算机软件程序和数据文件可以被访问、交换、拷贝和分发的简便也日益增长。为了利用这些计算机和通信系统以及它们提供的效率，存在对用于安全地存储和交换计算机软件与数据的方法的需要。

[0003] 已经证明广泛使用和接受的维持机密性或隐私的一种方法是使用秘密密码密钥对数据加密。现有的加密系统被设计为保护它们的秘密密钥或其他秘密数据免受“黑盒攻击”。这是这样的情形，其中攻击者具有算法的知识并且可以检查来自算法的各种输入和输出，但是对于算法本身的执行没有可见性(诸如自适应选择的输入 / 输出攻击)。

[0004] 尽管依赖于黑盒模型的密码系统是非常普遍的，但是已经表明该模型并未反映现实。通常，攻击者处于观察算法的执行的至少某个方面的位置，并且具有对目标算法的足够访问以挂载成功的攻击(即，信道侧攻击，诸如定时分析、功率分析、高速缓存攻击、故障注入等。)这样的攻击通常称为“灰盒”攻击，所假设的是攻击者能够观察系统执行的至少部分。

[0005] 认识到此，已经做出努力来设计抵抗更强大的攻击模型——“白盒攻击”的加密算法和数据信道。白盒攻击是对软件算法的攻击，其中假设攻击者对于算法的执行具有完全可见性。迄今为止，这样的保护系统已经经历了合理的成功，但是因为这样的保护系统变得越来越精密，所以攻击技术也越来越精密(诸如编码减少攻击、统计分桶(bucketing)攻击以及同态映射攻击)。因此，许多现有的白盒保护系统正被表明对于抵抗联合攻击是无效的。

[0006] 借助于简单编码的软件混淆有时已经在使用。为了是有用的，这样的编码在软件混淆中的应用必须不会过度增加软件的时间和空间消耗，因此这样的编码典型地相对简单。因此，尽管它们可以整体地保护软件，但是它们未提供高度的安全性。在软件中存在许多通信边界，所述边界代表具体的脆弱性：未受保护形式的数据进或出混淆程序的通道、数据进或出软件或硬件中的密码实现的通道等等。现有编码策略的强度典型地受它们保护的数据大小的严重限制。对于常规编码，这样的受保护的项大约为 32 比特，有时为 64 比特，并且有时是诸如字符或字节的更小数据段。鉴于编码和操作数大小的限制，一般无法防止对这样的编码的相当迅速的暴力破解。

[0007] 因此，存在对更有效的秘密隐藏和防篡改技术的需要，从而提供对软件代码和数据的总体保护以及对秘密秘密密钥、生物计量数据、加密的数据等的保护。还期望提供比常规简单编码强得多形式的对软件边界的保护。

发明内容

[0008] 本发明的实施例总地旨在提供更有效的秘密隐藏和防篡改技术，从而提供对软件代码和数据的保护，而不用担心安全性将被破坏。

[0009] 本文公开的方法和系统不限于任何特定基本程序。它们可以应用于密码系统，但是等同地，可以应用于非密码系统。被保护的代码也不规定保护它做了写什么，所以保护技术不受基本代码的约束。相对于可能留下或创建基于基本代码的模式的其他保护技术，这可以提供益处。这样的模式可能提供可被攻击者利用的弱点。

[0010] 本文公开的一些实施例提供“深刻数据相关性”，这可以使得难以或不可能排解或者区分受保护的代码和正在提供保护的代码。例如，AES 算法典型地始终执行相同方式，而不管是何输入数据。这使得攻击者直接直到他在找什么以及去哪里找到它。大多数白盒保护系统具有刚性等式结构，其未解决该类型的问题。即，攻击者可以知道找什么类型的操作或效果，以及何时在代码或执行中进行查看以找到那些操作或效果。相对地，本文公开的实施例可以提供非刚性的编码，诸如其中，保护算法的每次迭代导致不同的编码。因此，系统是极其不可重复的。除了别的以外，这可以使本文公开的实施例对于“比较”类型攻击是更有抵抗力的，在该“比较”类型攻击中，攻击者改变 1 比特并观察目标程序如何改变。以本文公开的一些实施例中，如果攻击者改变 1 比特，则受保护的代码将看起来完全不同。

[0011] 作为概述，本文描述的工具、一系列工具以及技术的实施例可以分组如下：

1)用于模糊目标代码模块之间以及目标代码与保护代码之间的边界的系统和技术。这例如可以通过将代码与周围代码掺合在一起以及将密码与其他代码进行交织来实现，这在其他保护性系统中通常未完成。

[0012] 2)用于确保破裂需要人干预的系统和技术。人寻找它们之前已经看到的模式。通过根据本文公开的实施例引入随机函数，可以移除重复性和 / 或常见模式，使得自动攻击是很大程度上无效的。

[0013] 3)用于保护防范“比较攻击”的系统和技术。如上所提及的，比较攻击是其中比较代码执行的两次迭代以参看差异的攻击，诸如改变单个输入比特以查看操作和输出如何改变。如本文所公开的保护算法可以以受保护代码的每次迭代产生动态不同的函数，所以比较攻击未提供任何有用信息。

[0014] 本文描述的混淆技术可以在开销可以被接纳的任何时候实现。白盒保护系统典型地比本文描述的技术具有更大开销，并且因此处于劣势。

[0015] 一些实施例包括用于软件保护的系统和技术，其通过向目标代码应用双射“基函数”来操作。这些基函数是互逆函数对f_K, f_K^{-1}，其例如用于对操作编码，并然后在软件应用中的稍后点处不对操作编码。编码使原始函数和其生成的数据模糊。信息没有损失，因为不编码操作容纳编码操作，从而在经编码的应用中稍后“撤销”或“反转”其影响。可以选择基函数对使得攻击者不能容易找到或确定逆函数。即，给定函数f_K，在没有密钥 K 的情况下可能不容易找到逆f_K^{-1}。密钥 K 可以在代码生成时使用，但是任何一旦函数f_K, f_K^{-1}已经生成并应用于目标代码就被丢弃。这些基函数对也是无损的，即，数学上可逆的。受保护的软件应用完全不需要对函数或过程进行解码以在目标代码中的其他位置使用它，因为编码和不编码改变包括在经编码的应用内。在一些实施例中，可能优选的是，基函数是“深度非线

性的”，因此使得同态攻击更加可能。在一些实施例中，基函数对可以包括置换多项式编码。置换多项式是可逆的多项式（多项式双射）。

[0016] 一些实施例可以生成或使用基函数对，以使得它们生成“实例多样性”和“动态多样性”。为了实现“实例多样性”，每个基函数对可以创建安全“通信信道”，诸如在软件应用的部分之间、在两个软件应用或平台之间等等。动态多样性可以通过将软件的操作链接到输入数据来创建。每当诸如针对在两个经编码的应用之间的通信执行编码时，可以在两个应用之间生成实例和动态多样性。基函数可以是高度“文本相关的”，所以它们提供良好的对明文和扰动攻击的抵抗力。如果攻击者改变任何内容，甚至是做出诸如 1 比特值的非常小的改变，则该改变将导致非常大的行为改变。该特征与常规密码代码形成鲜明对照，所述常规密码代码典型地对于代码的每次迭代产生相同的模式和结构，而不管攻击者做出的改变。通过做出小改变并观察影响，攻击者能够收集关于密码代码的操作的信息，但是他不能对使用本文公开的系统和技术编码的软件进行相同操作。本文公开的实施例提供的多样性还提供对“类攻击”的抵抗力。即，在所有情况下不可能提供这样的攻击方法，该攻击方法可以系统地并且自动地破裂本发明的每个实施例。还注意，常规白盒实现和代码优化器将不提供足够的多样性来获得任何有效保护。

[0017] 逆基函数的多样性和不可逆性极大地增加了攻击问题的复杂度。与常规软件代码或代码保护系统相对地，当尝试击败本文公开的系统和技术时，攻击者必须首先算出他正在攻击什么函数、代码部分、应用等等，然后如何对它求逆，以及然后如何利用它。

[0018] 本文公开的实施例提供的多样性可以提供可变、随机选择的结构来保护代码。生成基函数对和编码的引擎可以依赖于随机或伪随机密钥来选择基本函数和 / 或密钥。然而，根据本文公开的实施例的密钥不可以与许多常规安全系统的密钥一样大（即，64 或 128 比特）；相反，其可以是数千或数万比特。例如，开发的原型使用 2000 比特。

[0019] 本文公开的基函数可以包括用于编码、解码或重编码数据的双射。这样的双射可以包括以下特性：

1）对宽数据元素编码（典型地，四个或更多主计算机字宽）与典型标量编码（参见附录中列出的 [5,7]）不同，但是与块密码类似。

[0020] 2）仅对数据编码：与典型标量编码不同，但是与密码类似，它们不需要保护除在它们自己的对数据元素的重编码中涉及的那些以外的计算。

[0021] 3）取消数据块或流，和 / 或产生数据块或流的固定长度哈希以用于认证目的，类似于块密码，但是与标量编码不同。

[0022] 4）采用从软件的操作指令系统特意选择的操作的形式，所述操作将驻留在所述软件中并且所述操作将与所述软件互锁；即，它们被设计为类似在它们被嵌入的上下文中的代码，但是与密码不同。

[0023] 5）与密码和标量编码二者不同，采用海量多编码。标量编码一般采用一个或者至多几个数学构造。密码典型地采用稍大些的数量，但是该数量仍然很小。在本发明的一些实施例中，各种编码被应用于整个函数，从而创建产生自许多形式的保护相互交互的错综复杂交织的结构。

[0024] 6）与密码和标量编码二者不同，提供海量多样的算法结构。实施例可以没有固定数量的轮，没有用于各种子步骤的操作数的固定宽度，没有各种子步骤的固定互连，以及没

有任何种类的预定数量的迭代。

[0025] 7）与密码和标量编码二者不同，借助于高度数据相关算法来提供海量动态多样性：即，对于对基函数双射的任何具体采用，通过其子步骤的路径、其迭代计数等等强烈取决于要编码、解码或重编码的实际数据输入。

[0026] 8）与密码和标量编码二者不同，提供与它们的嵌入上下文海量互依性；即，它们的行为可以强烈取决于它们被嵌入其中的软件，并且可以使它们被嵌入其中的软件强烈取决于它们。

[0027] 一些实施例可以使用大的量的实际熵（即，大的真随机输入）。然而，如果生成基函数对的引擎自身未曝露于攻击者，则其可以安全地采用显著更小的密钥，该显著更小的密钥然后借助于伪随机数生成器生成大得多的伪随机密钥，因为在该情况中，攻击者必须应付实际密钥熵（其用于至伪随机数生成器的种子）和从该生成器的编程不可避免地产生的随机性。

[0028] 在一些实施例中，还可以使用偏置置换。如果内部数据用于生成基函数对或其他编码数据 / 函数而非随机数，则产生的编码将包含偏置。如果引入代码以创建编码可能容易明显的未偏置置换，则导致系统中的弱点。相对地，本文公开的实施例可以生成偏置置换，但是然后使用各种工具来使它们较少偏置。该方法已经表明比已知技术要不明显得多。

[0029] 一些实施例可以包括用于以下的技术：绑定管道开始和管道结束，使得目标软件代码在两端被系于应用或平台。例如，在对等数据传送环境或数字权利管理（DRM）环境中，这可以是有用的。本文公开的系统和技术还可以用于将密码系于其他软件应用或平台，这使用常规技术一般是难以完成的。

[0030] 一些实施例可以使用“函数索引的交织”。该技术提供来自线性部件和非线性等式求解的深非线性度。其可以以许多方式来使用，诸如边界保护、动态常数生成（例如，密钥到代码）、提供动态多样性（数据相关功能性）、自组合密码、密码混合以及组合密码与非密码。例如，其可以用于将黑盒密码与本文公开的其他保护代码进行混合，从而提供黑盒密码的长期安全性，具有白盒安全性的其他益处。如上文所提到的，本文公开的实施例的编码可以与运行时数据高度相关。以函数索引的交织，两种信息被使用：密钥 K，其确定基函数和结构，以及 R，其确定哪些混淆要应用于“定义实现”。典型地，客户端看不到 R。密钥 K 可以从上下文扩充，但是在本文描述的一些示例中，仅 R 以该方式扩充。可选地，来自用户或他的设备（诸如智能电话、平板计算机、PDA、服务器或台式计算机系统等等）的半一致信息或数据（诸如 IP 地址）可以用于如运行时密钥那样编码和解码。

[0031] 也可以使用递归函数索引的交织。函数索引的交织典型地对任意函数进行交织。如果这些函数中的一些是通过函数索引的交织获得的它们自身的函数，则这是函数索引的交织的递归使用。

[0032] 一些实施例可以包括随机交叉链接、交叉俘获、数据流复制、随机交叉连接以及随机检查，与代码重定序相组合，创建全向交叉相关性以及变量相关编码。

[0033] 一些实施例可以使用与分式变换（动态数据识别编码）的存储器混洗以隐藏数据流，这也可以被采用。在动态数据识别编码中，可以使用存储器单元的阵列 A，其可以视为具有虚拟索引 $0, 1, 2, \ldots, M-1$，其中 M 是阵列的大小和在有限环 $\mathbb{Z}/(M)$ 上的置换多项式 p 的模数（即，整数模 M），如在 C 程序阵列中那样。然而，对于任何给定索引 i，在阵列中存在其对

应于的不固定的位置，因为其被寻址为p(i)，并且p采用从至程序的输入确定的系数。位置$A[p(0)], A[p(1)], \ldots, A[p(M-1)]$可以被认为是外延主机器的那些的“伪寄存器”$R_1, \ldots, R_{M-1}$。通过将数据移进和移出这些寄存器、对每次移动时对移动的数据重编码以及通过针对许多不同值重用这些“伪寄存器”(例如，通过采用图着色寄存器分配)，攻击者跟随程序的数据流的难度可以大大增加。

[0034] 一些实施例可以使用“散布和掺合”编码。这是描述基函数加代码交织的使用的另一种方式，其“涂覆”基函数的边界以使得攻击者更加难以辨别它们。一般的数据掺合可以具有基函数的与其他代码混合的部分，使得更加难以标识和提升代码。

[0035] 一些实施例提供安全生命周期管理。黑盒安全性提供良好的长期保护，但是在今天的应用中不太有用。本文公开的实施例可以比实现在未受保护的设备上可能被破裂更快地刷新所述实现。不同的设备和应用具有不同的需要。例如，按次计费的电视广播(诸如体育赛事)在该赛事几天后可能具有很小的价值，所以可能仅有必要在一天左右提供足够安全性来保护该广播数据。类似地，计算机游戏市场可能在几周之后快速变小，所以可能关键仅在前几周或几月保护该游戏。本文公开的实施例可以允许用户应用需要的安全级别，从而权衡安全性与性能。字面上，可调整的“混淆刻度盘”可以放置在控制台上。尽管实现的特定定义的安全级别可能是未知的，但是混淆方法被应用的强度可以被控制。一般地，这些设定可以在应用被创建时用其嵌入的基函数进行调整，作为软件开发过程的部分。安全性分析可以提供对给定特定混淆级别破裂应用将有多困难的估计。基于该估计，可以做出如何平衡性能需要与安全性需要的工程决定，并且“混淆刻度盘”可以被相应地设置。这种灵活性是用其他保护系统无法获得的。例如，利用ASE，使用固定的密钥长度和固定的代码，它们不能被调整。

[0036] 一些实施例可以提供灵活的安全性刷新速率，允许对刷新代码的“移动目标”的复杂度的权衡。在许多情况中，所需的是足够快地刷新以保持领先于潜在攻击者。

[0037] 一些实施例可能没有在黑客曝露的环境中提供长期数据安全的主要目标。对此，方案是不将数据曝露给黑客，而是通过例如为客户端提供用于证书保护(安全ID(TM)，口令短语等)的web呈现而仅曝露访问该数据的手段，所述客户端经由受保护的对话来访问数据，所述对话可以至多曝露一小部分数据。在黑客曝露环境中，可以预期将部署以相同方式对曝露的软件进行刷新的过程。例如，在卫星TV条件访问系统中，嵌入在机顶盒(STB)中的软件中的密码密钥被定期刷新，使得任何对密钥的损害具有仅用于有限时间段的值。当前，可以借助于软件混淆和/或白盒密码在该有限曝露时段上保护这样的密码密钥。

[0038] 然而，白盒密码已证明对于这样的攻击是脆弱的，所述攻击可能由具有分析可执行程序的专家知识的密码学尖端攻击者非常快速地执行，因为所采用的密码算法在存在的被最透彻地检查的算法当中，并且用于分析程序的工具也已经在近来变得非常精细。此外，密码具有特殊的计算属性，因为它们经常在未在计算中通常使用的算术域上定义：例如，AES在伽罗瓦域上定义，RSA公钥密码系统通过在极大模数上的模算术来定义，在比特运算上的3DES，查找表以及以复制比特外延的比特置换。

[0039] 实际上，对程序的精细分析已经创建了有时可以完全绕过密码分析的需要：代码提升攻击，由此攻击者简单地提取密码算法并且在没有进一步分析的情况下采用它(因为它毕竟是操作的软件段，尽管其可以被混淆)来破裂软件应用的功能性。

[0040] 一些实施例可以提供强得多的对攻击的短期抵抗。这样的保护可以适于其中需要抵抗的时间相对短的系统，因为借助于刷新驻留在曝露平台上的软件解决了长期安全性。这解决了特定空白的需求，其关注由以下创建的紧张点：高度精细的密码分析工具和知识、被极好研究的密码、经由软件混淆可提供的有限保护、用于分析可执行程序的高度精细工具，以及软件在典型商业内容分发环境中有限的曝露次数。目标是防止这些种类的攻击，使用白盒密码的经验已经示出为在现有技术内：快速密码攻击和/或代码提升攻击如此之快，以至于即使给出在曝露程序(诸如STB程序)的刷新之间有限的有效性生命跨度，它们也具有值。

[0041] 在许多情况中，仅有必要抵抗在刷新周期期间的持续时间内的分析，并且将密码替换如此紧密地系于其驻留在其中的应用，从而代码提升攻击在刷新周期的持续时间内也是不可行的。刷新周期率由工程和成本考虑来确定，有多少带宽可以分配给刷新，我们可以多平滑地将刷新与进行中的服务整合在一起而不损失服务质量等等：这些都是在提供条件访问系统领域中非常好理解的问题。这些考虑粗略地指示对于分析和提升攻击我们的保护必须坚持多久。

[0042] 一些实施例可以通过以下提供可以抵抗更长时间段内的攻击的显著更大的编码：放弃用经编码的操作数进行计算的观念——如上使用较简单的编码所做的——并用更像密码的事物来替换它。密码本身可以是并且确实用于该目的，但是它们常常不能容易地与普通软件互锁，因为(1)它们的算法被密码标准刚性地固定以及(2)它们的计算典型地与普通软件非常不同并因此既不容易在其内被隐匿也不容易与其互锁。本文描述的基函数提供了允许隐匿和互锁的替代：它们利用常规操作，并且它们的算法与用密码的情况相比极大地更加灵活。它们可以与密码组合来将与常规密码一样强的黑盒安全级别与显著优胜于如上的简单编码和已知的白盒密码二者的白盒安全级别进行组合。

[0043] 在一些实施例中，可以通过以下来创建基函数：选择字大小w和矢量长度N，并生成可逆状态矢量函数，所述可逆状态矢量函数被配置为对w元素字的N矢量进行操作，所述可逆状态矢量函数包括多个可逆操作的组合。所述状态矢量函数可以接收至少64比特的输入并提供至少64比特的输出。所述状态矢量函数中的步骤的第一部分在$Z/(2^w)$上执行线性或仿射计算。使用第一和第二编索引技术对所述状态矢量函数中的步骤的部分编索引。然后可以修改在现有计算机程序中的至少一个操作以取代所选择的操作而执行所述状态矢量函数。每个编索引技术可以控制不同的编索引操作，诸如if-then-else构造、switch、元素置换选择、迭代计数、元素旋转计数、函数索引的密钥索引等等。所述状态矢量函数中的步骤的一些可以是非T函数操作。一般地，所述状态矢量函数中的每个步骤可以可逆的，使得整个状态矢量函数是通过对每个步骤取逆而是可逆的。在一些配置中，所述状态矢量函数可以使用例如运行时密钥、生成时密钥或函数索引的密钥来加密钥。所述状态矢量函数可以通过各种操作类型来实现，诸如线性操作、矩阵操作、随机交换等。各种编码方案还可以应用于所述状态矢量函数的输入和/或输出，和/或所述状态矢量函数的操作。在一些配置中，不同的编码可以被应用以在与所述状态矢量函数相关联的各个点处产生分式。

[0044] 在一些实施例中，本文所公开的基函数可以通过例如以下来执行：接收具有字大小w的输入，向所述输入应用可逆状态矢量函数，所述可逆状态矢量函数被配置为对w元素字的N矢量进行操作，其中所述可逆状态矢量函数包括多个可逆操作，并且所述状态矢量

函数中的步骤的第一部分在$Z/(2^n)$上执行线性或仿射计算。可以向所述可逆状态矢量函数的输出应用额外操作，其中每一个操作是基于不同编索引技术而选择的。一般地，所述状态矢量函数可以具有本文关于状态矢量函数和基函数公开的任何属性。

[0045] 在一些实施例中，第一操作可以通过执行第二操作来执行，例如通过以下来执行：接收用第一编码 A 编码为 A(X) 的输入 X，使用$B^{-1}(X)$的值对所述输入执行第一多个计算机可执行操作，其中B^{-1}是第二编码机制 B 的逆，所述第二编码 B 不同于所述第一编码 A，基于$B^{-1}(X)$来提供输出。这样的操作可以认为是“分式”，并且可以允许操作被执行而对于外部用户或者对于潜在攻击者是不可访问或不可见的。在一些配置中，所述第一操作的输出未被在外部提供给可执行代码，所述第一操作与所述可执行代码被集成在一起。

[0046] 在一些实施例中，对于被配置为接收输入并提供输出的矩阵操作，在执行操作之前，可以根据排序网络拓扑来置换所述输入。可以使用置换后的输入执行所述矩阵操作以生成输出，并且根据所述排序网络拓扑来置换所述输出。可以将置换后的输出提供为所述矩阵操作的输出。

[0047] 在一些实施例中，可以接收第一输入，并且向所述第一输入应用函数索引的交织的第一函数以生成具有左部分和右部分的第一输出。可以向所述第一输出应用函数索引的交织的第二函数以生成第二输出，其中，所述第一输出的左部分被用作所述第二函数的右输入，并且所述第一输出的右部分被用作所述第二函数的左输入。然后可以提供所述第二输出作为对所述第一输出的编码。

[0048] 在一些实施例中，可以生成密钥 K，并且基于密钥 K 和随机化信息 R 来生成一对基函数f_K, f_K^{-1}。可以向通信管道的第一端应用基函数f_K，并且向所述通信管道的第二端应用逆，之后可以丢弃所述密钥 K。通信管道可以跨单个平台上或分离平台上的应用。

[0049] 在一些实施例中，要由计算机系统在程序执行期间执行的一个或多个操作可以被复制以创建一个或多个操作的第一拷贝。程序任何可以被修改为执行第一操作拷贝而非第一操作。每个操作和对应的拷贝可以使用不同编码来编码。操作对也可以用于创建检查值，诸如其中，在操作结果的执行与拷贝的执行之间的差异被添加到操作的结果或操作拷贝的结果。这可以允许检测攻击者在程序执行期间做出的修改。

[0050] 在一些实施例中，在包括多个操作和每个操作的拷贝的程序的执行期间，在到达在其处应当执行多个操作中的操作的执行点时，拷贝或原始操作可以被随机选择并由程序执行。随机选择的操作的结果可以与将在只有操作的单个拷贝被执行时获得的结果相等。

[0051] 在一些实施例中，可以从应用接收输入。可以用多个 M 寄存器位置$c_1,\ldots,c_n$来定义大小为 M 的阵列，$n \leq M$。还可以定义置换多项式 p、从输入产生 z 的基于输入的$1\times n$矢量映射矩阵 A，以及一系列常数$c_i = p(z+i)$。然后可以执行一系列操作，每个操作提供中间结果，所述中间结果存储在从 M 寄存器随机选择的 M 寄存器中。然后可以基于一系列中间结果将最后结果从存储最后结果的最后 M 存储器提供到应用。存储在 M 寄存器中的每个中间结果可以具有在将中间结果存储到对应 M 寄存器中之前应用于中间结果的单独编码。应用于中间结果的不同编码可以从多个不同编码当中随机地选择。类似地，可以或可以不对应于用于将中间结果存储在 M 寄存器中的编码的不同编码可以应用于存储在 M 寄存器中的中间结

果。新的 M 寄存器可以按照需要被分配，例如仅当根据图着色分配算法需要时被分配。

[0052] 在一些实施例中，产生至少第一值 a 作为输出的第一操作 $g(y)$ 可以被执行，并且使用 a 和第二值 b 将第一变量 x 编码为 aX+b。可以使用 aX+b 作为输入来执行第二操作 $f(aX+b)$，并且可以执行使用 a 和 b 的解码操作，之后 a 和 b 可以被丢弃。值 b 还可以是第三操作 $h(z)$ 的输出。使用 $g(y)$ 和 / 或 $h(z)$ 的不同执行实例，不同编码可以用于编码为 aX+b 的多个输入值。可以基于一个或多个常数被存储在存储器中的预期时间来从存储在计算机可读存储器中的任何值中选择这些值。类似地，包含用于执行操作 $f(aX+b)$ 和 $g(y)$ 的指令的现有计算机可读程序代码可以被修改为将 x 编码为 cX+d，并且当执行时 $g(y)$ 至少产生第一值 c。可以针对至少一个 x 执行操作 $f(cX+d)$，并且 c 和 d 后续被丢弃。

[0053] 在一些实施例中，至少一个基函数可以与用于现有应用的可执行程序代码掺合。例如，可以通过用基函数替换现有程序中的至少一个操作，来将基函数与可执行程序代码进行掺合。还可以通过应用本文所公开的技术中的一种、一些或全部来将基函数与现有应用进行掺合，所述技术包括分式、变量相关编码、动态数据识别编码和 / 或交叉链接。所使用的基函数和 / 或任何掺合技术可以包括或者可以排他地包括这样的操作，所述操作与它们与之掺合的现有应用程序代码的部分中呈现的操作是类似或不可区分的。因此，攻击者可能难以或不可能在没有基函数的情况下从将在现有可执行程序代码中呈现的那些中区分基函数和或混合技术操作。

[0054] 在一些实施例中，可以提供一种计算机系统和 / 或计算机程序产品，包括处理器和 / 或存储指令的计算机可读存储介质，所述指令使得所述处理器执行本文所公开的一种或多种技术。

[0055] 此外，因为本文公开的与基函数一起使用的算法可以是相对灵活的和开放式的，所以它们允许软件多样性的高度灵活的方案，并且改变的实例与使用白盒密码可能的情况相比更深地不同。因此，它们远不易受自动攻击的攻击。只要可以迫使攻击需要人参与，就是高度有利的，因为我们可以新建受保护代码的实例并且数据可以以计算机速度被自动生成，而仅可以以人类速度来损害它们。

[0056] 在考查以下附图和详细描述后，本发明的其他系统、方法、特征和优点对于本领域技术人员将是或者将变得清楚。意图将所有这样的附加系统、方法、特征和优点包括在本说明书内、在本发明的范围内以及受所附权利要求的保护。

附图说明

[0057] 在附图中：

图 1 示出了根据本发明的用于加密函数的交换图表；

图 2 示出了根据本发明的虚拟机一般指令格式；

图 3 示出了根据本发明的虚拟机进入 / 退出指令格式；

图 4 示出了根据本发明的标记 I“木人(Woodenman)”构建；

图 5 和 6 示出了根据本发明的标记 II 构建的分别的第一半和第二半；

图 7 示出了根据本发明的排序网络的图形表示；

图 8 示出了根据本发明的执行函数索引的交织的方法的流程图；

图 9 示出了根据本发明的执行控制流复制的方法的流程图；

图 10 示出了根据本发明的执行数据流复制的方法的流程图；

图 11 示出了根据本发明的创建f_k区段的方法的流程图；

图 12 给出了用于实现本发明的标记 II 保护系统的处理流程图；

图 13 示出了本发明的标记 III 实现的区段设计的不规则结构的图形表示；

图 14 示出了可以用本发明的标记 III 实现中的 T 函数划分实现的粒度的图形表示；

图 15 示出了本发明的标记 III 实现的总体结构的图形表示；

图 16 示出了本发明的标记 III 实现的防御层的图形表示；

图 17 示出了本发明的实现中的海量数据编码的图形表示；

图 18 和 19 示出了本发明的实现中的控制流编码的图形表示；

图 20 示出了本发明的实现中的动态数据识别编码(mangling)的图形表示；

图 21 示出了本发明的实现中的交叉链接和交叉俘获的图形表示；

图 22 示出了本发明的实现中的上下文相关编码的图形表示；

图 23 给出了用于实现本发明的标记 II 保护系统的处理流程图；

图 24 示出了在本发明的实现中的海量数据编码或动态数据识别编码的典型使用的图形表示；

图 25 示出了阐述本发明的实施例寻求解决的主要问题的示例性框图；

表 25 给出了对软件边界问题进行分类的表；

图 26 示出了未受保护形式的、白盒保护的以及用本发明的系统进行保护的示例性软件系统的框图；

图 27 示出了对比由黑盒安全性、白盒安全性以及在本发明的示例性实施例下的保护所提供的保护级别的柱状图；

图 28 示出了对比密码、哈希以及根据本发明的示例性基函数的处理流程图；

图 29 示出了本发明的基函数可以如何用于提供安全通信管道的示例性框图；

图 30 示出了根据本发明的用于函数索引的交织的处理流程图；

图 31 给出了用于实现本发明的标记 I 保护系统的处理流程图。

具体实施方式

[0058] 本文公开的实施例描述了可以允许保护计算机系统的可能暴露于攻击者的方面的系统、技术和接收机程序产品。例如，已经在供最终用户操作的商品硬件上分发的软件应用可能遭受在执行期间具有对代码的访问的实体的攻击。

[0059] 一般而言，本文公开的实施例提供用于创建基函数的集合以及以以下方式将那些函数与现有程序代码进行集成的技术：使得对于潜在攻击者而言难以或者不可能隔离、区分或者接近地检查基函数和 / 或现有的程序代码。例如，本文公开的处理可以接收现有程序代码并将基函数与现有代码进行组合。也可以使用诸如分式、动态数据识别编码、交叉链接和 / 或如本文所公开的变量相关编码的各种技术来组合基函数与现有代码，以进一步掺合基函数与现有代码。基函数和其他技术可以使用计算上与现有程序代码使用的那些类似、相同或不能区分的操作，这可以增加潜在攻击者区分受保护代码与应用的保护技术的难度。如本文将描述的，这可以提供这样的最终软件产品，其与使用常规保护技术可能的情

况下相比对于各种攻击是更有弹性的。

[0060] 如在图 25 中所示，本文公开的实施例可以提供用于若干基本问题的解决方案，这些基本问题在要保护软件免受攻击时出现，所示基本问题诸如软件边界保护、高级多样性和可再生性问题以及保护可测量性问题。

[0061] 软件边界问题可以组织成五组，如表 1 中所示：外皮问题、数据边界、代码边界、受保护数据与受保护代码之间的边界以及受保护软件与安全硬件之间的边界。

[0062] 表 1

边界问题		描述
外皮问题	数据从未受保护域流到受保护域	对未受保护数据和计算的攻击可以是损害它们在受保护域中的数据和计算对应物的起始点。在不在边界引入可信使能机制的情况下，这些问题典型地难以解决。
	数据从受保护域流到未受保护域	
	未受保护域与受保护与之间的计算边界	
数据边界	数据类型边界	当前数据变换技术受限于单独数据类型，非多个数据类型或海量数据。独特的受保护数据项当中的边界突出，从而允许标识和分区。
	数据相关边界	经由现有数据流保护的数据扩散受到限制。原始数据流和计算逻辑被暴露。大多数当前白盒密码的弱点与数据类型和数据相关边界问题有关。
	跨功能的数据边界	应用系统的功能部件当中的数据通信——无论在相同还是不同设备上运行或者作为客户端和服务器——都被致使是易受攻击的，因为通信边界是清楚易见的。
代码边界	受保护部件当中的功能边界	功能部件当中的边界在保护那些边界之后仍然是可见的。例如，白盒密码部件可以通过它们的独特计算来标识。一般地，这样的受保护计算区段可以被容易地分区，从而产生对于基于部件的攻击的易受攻击性，所述基于部件的攻击诸如代码提升、代码替换、代码克隆、重放、代码嗅探以及代码欺骗。
	注入的代码与原始应用代码的受保护版本之间的边界	当前的单独保护技术产生对于具体计算局部化的受保护代码。使用不同保护技术所产生的代码边界未被有效地粘合和互锁。

受保护数据与受保护代码之间的边界	受保护数据和受保护代码未被有效地锁定在一起来防止代码或数据提升攻击。当前的白盒密码实现对于本领域中的这样的提升攻击是易受攻击的。
受保护软件与安全硬件之间的边界	我们缺少有效的技术将安全硬件和受保护软件相互锁定。受保护软件与安全硬件之间的边界是易受攻击的，因为跨边界的数据未受保护或者受到弱保护。

[0063] 存在三种类型的“外皮问题”可以通过本文公开的实施例来解决：从未受保护到受保护域的数据流、从受保护到未受保护域的数据流，以及未受保护与受保护域之间的计算边界。最终，数据和用户交互应当以未编码形式来执行，使得用户可以理解该信息。在每种情况中，对未受保护数据和计算的攻击可能是损害受保护域中的它们的数据和计算对应物的开始点。常规地，在不在边界处引入可信使能机制的情况下，这些问题很难解决。然而，本文公开的实施例提供的多样性以及在边界自身处的编码提供了已知系统未提供的保护程度。

[0064] 数据边界可以被分类为三种类型之一：数据类型边界、数据依赖性边界以及跨功能部件的数据边界。关于数据类型边界，当前的数据变换技术受限于单独的数据类型而非多种数据类型或海量数据。独特受保护数据项当中的边界突出，从而允许标识和分区。关于数据依赖性边界，经由现有数据流保护的数据扩散受限制：原始数据流和计算逻辑被暴露。大多数当前的白盒密码弱点与数据类型和数据依赖性边界问题二者相关。最后，关于跨功能部件的数据边界，应用系统的功能部件当中的数据通信——无论在相同还是不同设备上运行或者作为客户端和服务器——都被致使是易受攻击的，因为通信边界是清楚易见的。本文公开的实施例对基函数编码和函数索引交织的使用可以这些数据边界问题的一些或全部，因为数据和边界自身二者都可以是模糊的。

[0065] 代码边界可以被分类成两种类型：受保护部件当中的功能边界以及注入代码与原始应用代码的受保护版本之间的边界。受保护部件当中的功能边界是弱点，因为功能部件当中的边界在保护那些部件之后仍然是可见的。即，用白盒保护，白盒密码部件可以一般地通过它们的独特计算来标识。一般地，这样的受保护计算区段可以被容易地分区，从而产生对于基于部件的攻击的易受攻击性，所述基于部件的攻击诸如代码提升、代码替换、代码克隆、重放、代码嗅探以及代码欺骗。类似地，注入保护代码与原始应用代码的受保护版本之间的边界一般也是可见的。当前的单独保护技术产生对于具体计算局部化的受保护代码。使用不同保护技术所产生的代码边界未被有效地粘合和互锁。相对地，本文公开的实施例对基函数编码和函数索引交织的使用可以解决所有这些代码边界问题，因为代码可以是模糊的并且与保护代码自身进行交织。因为基本计算机处理和算术函数用于保护代码，所以不存在攻击者将快速标识的独特代码。

[0066] 受保护数据与受保护代码之间的边界呈现另一弱点，该弱点可能被攻击者利用，

因为当前的白盒技术不保护受保护数据与受保护代码之间的边界。相对地，本文公开的实施例可以将受保护数据与受保护代码锁定在一起，从而防止代码或数据提升攻击。当前的白盒密码实现对于本领域中的这样的提升攻击是易受这攻击的。

[0067] 类似地，受保护软件与安全硬件之间的边界呈现脆弱性，因为现有的白盒技术不保护受保护软件与安全硬件之间的边界——跨这样的边界的数据不受保护或者受弱保护。相对地，本文公开的实施例可以将受保护硬件和受保护软件相对于彼此进行锁定。

[0068] 还存在与安全性相关联的组织(logistical)问题，具体地，多样性和可再生性问题。当前的程序多样性受程序构造和结构的限制，并且受应用的单独保护技术的局限性的限制。结果，多样化的实例并未深度改变(例如，程序结构变化及其有限)，并且实例可能足够类似，从而允许基于比较多样化的实例的攻击。当前的保护技术受限于静态多样性和固定安全性。相对地，本文所公开的实施例可以提供动态多样性，其可以允许对多样性和可再生性提供的安全性级别的智能控制和管理。如本文进一步详细公开的，解决高级多样性和可再生性问题对于安全性生命周期管理而言可以是根本的。

[0069] 图 26 示出了在已知白盒模型下以及在本文所公开的示例实施例下保护的示例软件系统的框图。要保护的原始代码和数据功能、模块以及存储块通过标记为 F1、F2、F3、D1 和 D2 的几何形状来表示。现有的白盒和类似的保护技术可以用于保护各种代码和数据功能、模块以及存储块，但是即使在受保护形式下，它们也将(至少最低限度地)公开在它们边界处的未受保护数据和其他信息。相对地，本发明的实施例可以解决这些边界问题。在一些情况中，一旦如本文所公开的实施例的实例已经被执行，观察者就不能根据原始程序讲出哪些部分是 F1、F2、F3、D1、D2 和数据，即使观察者具有对程序的访问并且可以观察和改变其操作也是如此。

[0070] 这可以例如通过在不同代码和数据功能、模块以及存储块之间将代码交织在一起来实现，由此将这些部件“粘合”在一起。以这种方式使代码紧密联系，可以提供真正的边界保护。如上文所描述的，多样性和可再生性在以下方面提供 ：1) 比过去的系统提供大得多的灵活性 ；2) 容易且强有力的控制 ；3) 实现动态多样性和安全性 ；以及 4) 可测量和可管理的多样性。本文公开的实施例还可以提供单向双射函数的“复杂性属性”以及可测量、可控制和可审计机制，以保证用户的所需安全性。后文更详细地描述了双射，但是简言之，它们是无损函数对f_K, f_K^{-1}，其执行函数的变换，该变换稍后在受保护代码中被撤销。该变换可以用数千或数百万种方式来完成，每种变换一般以完全不同且不可重复的方式来完成。各种技术可以用于隐藏现有程序，从而实现双射函数的大量多编码(multicoding)，其不是人工编程的而是由随机计算过程来生成的。这包括可以用像密码和哈希这样的方式用来解决边界问题的双射函数。

[0071] 相对于常规技术，本文公开的实施例可以提供改进的安全性和安全性保证(即，有效的安全性和有效的安全性度量)。还可以实现比白盒密码所提供的更大的时间和空间上的多样性。安全性度量基于已知攻击的计算复杂性，基本原语是互逆函数对的生成。其他原语可以用或不用对称或非对称辅助密钥如本文所描述的那样来构造。

[0072] 图 27 在长期安全性和抗敌对攻击方面将常规黑盒和白盒模型与本文公开的实施例的属性进行对比。密码术很大程度上依赖于密码和哈希 ；密码使得秘密能够在不安全或

公共信道上传送，而哈希使出处有效。这些能力具有庞大数量的使用。在黑盒环境中，这样的密码技术可以具有非常好的长期安全性。然而，在抗攻击方面，这样的系统具有非常短的寿命。如上文所解释的，密码和哈希具有严格的结构和非常标准化的方程，它们直接受到攻击。白盒保护可以用于改进抗攻击的级别，但是即使在这样的环境中，受保护代码仍将揭露来自原始密码代码和哈希代码的模式和方程，并且边界将不受保护。此外，白盒保护将不提供保护代码免受扰乱（perturbation）攻击。

[0073] 相对地，本文公开的实施例可以并入像密码和像哈希那样的编码，这向保护编码给与密码和哈希的安全性和强度。换言之，将白盒编码应用于密码和哈希的过程典型地在尝试保护和模糊非常独特代码中使用简单的编码。然而，本文公开的技术可以使用强的、多样的编码来保护任何代码。利用所公开的多样编码和交织，在目标代码中的独特性将被移除。因此，如所示出的，所公开的技术可以提供比常规黑盒和白盒保护强得多的保密分布图（security profile）。

[0074] 图 1 示出了根据本发明的实施例的使用编码的加密函数的交换图表。对于 F，其中 $F:D\rightarrow R$ 是总计，双射 $d:D\rightarrow D'$ 和双射 $r:R\rightarrow R'$ 可以被选择。$F'=r\circ F\circ d^{-1}$ 是 F 的编码版本；d 是输入编码或域编码，并且 r 是输出编码或范围编码。诸如 d 或 r 的双射被简称为编码。在其中 F 是函数的特定情况中，在图 1 中示出的图表然后交换，并且利用 F' 的计算是利用加密函数的计算。关于这样的编码的使用的附加细节在附录的 2.3 节总体提供。

[0075] 图 28 将常规密码和哈希的属性与本文公开的双射基函数的那些属性进行对比。密码是无损函数；它们保存它们编码的所有信息，因此所述信息可以未被编码并且以与原始相同的方式被使用。密码被可逆地提供，一方被给予一个或多个密钥，但是难以从明文和加密的信息（图 28 中的“明文”和“加密的”）的实例确定该一个或多个密钥 *K1*、*K2*。哈希是在某个长度之上有损的，但是这通常不是问题，因为哈希一般仅用于验证。利用哈希，难以从原始数据和哈希（图 28 中的“明文”和“哈希的”）的实例确定可选密钥 K。

[0076] 本文公开的基函数可以替代密码或哈希提供服务，因为难以从编码和未编码函数 f_K、f_K^{-1}。相对于使用密码或哈希，基函数提供的优点在于基函数使用的计算与普通代码更类似，这使得更容易将基函数的代码与目标代码进行掺合。如上文所提到的，密码和哈希使用难以模糊或隐藏的非常独特的代码和结构，而导致脆弱性。

[0077] 如本文所公开的互逆基函数对可以以以下两种方式来采用随机秘密信息（熵）：作为用于确定互逆函数 f_K、f_K^{-1} 的密钥信息 K，以及作为确定如何模糊 f_K、f_K^{-1} 实现的随机化信息 R。

[0078] 例如，两个互逆基函数可以由子例程 G 和 H 表示，写为 C。基函数可以通过自动化的基函数生成器程序或系统来构造，其中 G 是数学函数 f_K 的混淆实现，并且 H 是数学函数 f_K^{-1} 的混淆实现。因此，G 可以用于“加密”数据或代码，然后其可以用 H 来“解密”（或者反过来）。

[0079] 可选地，除了建立时密钥 K 之外，还可以提供运行时密钥。例如，如果给定基函数的输入比输出宽，则额外输入矢量元素可以用作运行时密钥。这与具有诸如 AES-128 的密码的情形非常相像。AES-128 的典型运行具有两个输入：一个是 128 比特密钥，一个是 128

比特文本。该实现在密钥的控制下执行加密或解密。类似地，基函数可以构造为取决于其额外输入的内容来不同地加密，使得实际上的额外输入成为运行时密钥（与控制基函数的静态方面的软件生成时间密钥 K 相反）。本文公开的基函数的建立块使得相对容易地指示运行时密钥对于 f_K, f_K^{-1} 二者的实现是否是相同的，或者对于 f_K 和对于 f_K^{-1} 是不同的；如果将运行时密钥添加到选择器矢量，则其对于 f_K 和 f_K^{-1} 是相同的，而如同运行时密钥添加到其他地方，则其在 f_K 和 f_K^{-1} 之间不同。

[0080] 与已知白盒系统中相比，密钥信息 K 可以用于选择不同得多的编码函数，从而允许强得多的空间和时间多样性。多样性还与在本发明的实施例中使用的其他技术一起被提供，诸如函数索引交织，其经由文本相关来提供动态多样性。还可以通过后文中描述的控制流编码和海量数据编码的变体来提供进一步的多样性。

[0081] 本文所公开的基函数可以并入或利用状态矢量函数。一般而言，如本文所使用的，状态矢量函数围绕 N 个元素的矢量来组织，其每个元素是 w 比特的量。状态矢量函数可以使用一系列步骤来执行，在每个步骤中，矢量的介于零与 N 之间的数量的元素被修改。在其中修改零个元素的步骤中，该步骤实质上是对状态矢量应用恒等函数。

[0082] 在一些实施例中，在构造基函数中使用的一个或多个状态矢量函数可以是可逆的。如果对于状态矢量函数中的每个和所有步骤，存在步骤逆，使得应用步骤算法及随后应用步骤逆算法没有净效果，则状态矢量函数是可逆的。通过以它们的原始的相反顺序执行逆步骤算法，任何有限序列的可逆步骤是可逆的。

[0083] W 比特元素的矢量上的可逆步骤的说明性示例包括：将两个元素相加，诸如将 i 和 j 相加以获得 $i+j$；将元素乘以在 $Z/(2^w)$ 上的奇常量；通过取与 $Z/(2^w)$ 上的可逆矩阵的乘积来将元素的连续或非连续子矢量映射到新值。这些示例的相关联的逆步骤分别是：从元素 j 减去元素 i；将元素乘以 $Z/(2^w)$ 上的原始常量乘数的乘法逆元；以及通过乘以所述矩阵的逆来将子矢量映射到其原始值。

[0084] 一些实施例可以使用具有一个或多个索引步骤的一个或多个状态矢量函数。如果除了其正常输入之外，其还取额外的索引输入使得改变索引就改变计算函数，则步骤被索引。例如，可以通过常量矢量对添加常量矢量的步骤进行索引，或者可以通过应用的置换来对置换子矢量的步骤进行索引。在每种情况中，至少部分地通过提供给函数的索引来确定所执行的特定函数。

[0085] 索引的步骤还可以是可逆的。一般地，如果为每个索引计算可逆步骤，并且用于计算所述步骤的索引或者从其可以获取所述索引的信息在对所述步骤求逆时是可用的，则索引的步骤是可逆的。例如，如果 S_{17}^{-1} 被定义，并且索引（17）在合适的时间是可用的以确保其 S_{17}^{-1} 在对状态矢量函数求逆时被计算，则 S_{17} 是可逆的。作为示例，步骤可以对所述状态的一些元素进行操作。为了对该步骤进行索引，所述状态的其他元素可以用于计算所述索引。如果随后对其他元素执行可逆的步骤，则可以通过对那些步骤求逆来获取索引，只要两个元素集合不重叠即可。

[0086] 如本文所公开的函数索引的交织是基函数内索引步骤的使用的原理的特定示例。

如本文所公开的索引步骤的其他使用可以包括：允许加密钥的状态矢量函数的创建：在一些索引步骤中使用的索引集合可以用作密钥。在该情况中，索引不是从计算内获得的，而是由额外输入提供的；即，函数取状态矢量加上密钥作为输入。如果索引步骤是可逆的并且普通的非索引步骤是可逆的，则整个状态矢量函数是可逆的，相当像密钥密码。

[0087] 在一些实施例中个，索引信息可以提供或者用作所生成的基函数的密钥。如果当状态矢量函数生成时针对索引信息来部分地评估状态矢量函数，使得索引在所生成的函数的执行中不显式地出现，则其是生成时密钥。如果在状态矢量函数的执行期间生成用于处理索引信息的代码，使得在所生成的函数的执行中显式地出现，则其是运行时密钥。如果代码在内部生成在状态矢量函数内的索引，则其是函数索引的密钥。

[0088] 在实施例中，基函数可以基于初始选择或标识的字大小 w 来构造。在一些配置中，主机平台的默认整数大小可以用作字大小 w。例如，在现代个人计算机上，默认整数大小典型地为 32 比特。作为另一示例，可以使用例如在 C 中所使用的短整数长度，诸如 16 比特。在其他配置中，可以使用 64 比特的字大小。还选择用于基函数的矢量长度 N，其表示 w 大小的字中输入和输出的长度，典型地在内部包含四个或更多个字。在一些实施例中，诸如使用如本文所公开的交织技术的情况下，字大小 w 可以优选地为 N 矢量的内部字大小的两倍。状态矢量函数然后可以通过级连一系列步骤或者步骤的组合来创建，每个步骤对 w 元素字的 N 矢量执行可逆步骤。状态矢量函数的逆可以通过以相反顺序级连步骤的逆来生成。

[0089] 在一些实施例中，一个或多个密钥还可以并入状态矢量函数中。各种类型的密钥可以应用于状态矢量函数或者与状态矢量函数集成在一起，包括如之前所描述的运行时密钥、生成时密钥以及函数索引密钥。为了生成加运行时密钥的状态矢量函数，该函数可以被修改为显式地接收密钥作为至函数的额外输入。为了生成加生成时密钥的状态矢量函数，在状态矢量函数中的代码可以针对所提供的密钥被部分地评估。对于许多类型的操作，这样单独的情况或者结合典型编译器优化的情况可以足以使密钥在所生成的代码内是不可恢复的或不明显的。为了生成加函数索引密钥的状态矢量函数，可以将状态矢量函数构造为使得按照状态矢量函数内的需要来提供用于逆操作的合适密钥。

[0090] 在一些实施例中，可以优选地选择用于状态矢量函数的这样的实现，所述实现接受相对宽的输入并提供相对宽的输出，并且所述实现包括可逆步骤的完备集合。特别地，可以优选地构造这样的实现，所述实现接受至少 64 比特的宽输入和输出。对于在状态矢量函数中的大量步骤，诸如至少 50％或更多，可以优选地为在 $Z/(2w)$ 上的线性或仿射操作。还可以优选地为状态矢量函数选择具有宽泛多样化的步骤。

[0091] 在一些实施例中，可以优选地使用多种形式的索引对诸如至少 50％或更多的很大部分的步骤进行索引。索引的合适形式包括 if-then-else 或 switch 构造、元素置换选择、迭代计数、元素旋转计数等等。对于索引的一些或全部，还可以优选地为如本文所公开的函数索引密钥。

[0092] 在一些实施例中个，对于状态矢量函数的初始和 / 或最后步骤，可以优选地是跨整个状态矢量混合输入熵的步骤，典型地，所述输入熵不同于任何单独的密钥输入。

[0093] 在一些实施例中个，可以优选地将状态矢量函数构造为使得至少每几个步骤就执行一个非 T 函数步骤。参照编程操作，T 函数步骤的示例包括加法、减法、乘法、按比特 AND、按比特 XOR、按比特 NOT 等等；非 T 函数步骤的示例包括除法、模赋值（modulo

assignment)、按比特右移位赋值(assignment)等等。非 T 函数步骤的其他示例包括加函数索引密钥的逐元素旋转、子矢量置换等等。如之前所公开的,非 T 函数步骤的包括可以防止或者降低诸如比特切片(bit-slice)攻击的某些类型的攻击的有效性。

[0094] 如之前所描述的,状态矢量函数对包括如本文所描述的状态矢量函数和状态矢量函数的完全逆。在操作中,状态矢量函数对的构造可以但是并不一定通过例如以下方式执行:以诸如 C++ 代码等的语言源的形式将一系列参数化算法和/或逆算法进行组合。类似地,生成时密钥的替换可以但是并不一定通过以下方式来执行:宏预处理器中的宏替换、函数内嵌(function in-lining)以及参数化模版的使用的组合。这样的组合、替换和其他操作可以在如本文所公开的状态示例生成系统内被自动化。一旦已经生成了状态矢量函数对,可以使用二进制和/或编译器级工具进一步修改所生成的代码来保护其中的一者或两者。在一些实施例中,对状态矢量函数对中的一个或两个函数做出的特定修改可以基于每个成员是否被预期在很可能遭受攻击的环境中执行来选择。

[0095] 例如,在一些实施例中,被预期处于暴露环境中的函数或函数的一部分可以被限界在输入矢量被提供到状态矢量函数所在的点附近,和/或在其中输出矢量被其调用代码消耗的点附近。代码可以通过例如使用如本文所公开的动态数据识别编码和/或断裂来限界。例如,所提供的输入可以来自编码识别的储存器,并且输出可以通过调用器从编码识别的储存器取得。其他技术可以用于在这些点处绑定代码,诸如本文所公开的具有交叉链接和交叉俘获的数据流复制。可以使用不同的组合,诸如其中动态数据编码识别、断裂和数据流复制在相同点都被应用以在所述点处绑定代码。预期在暴露环境中应用于代码的保护可以在状态矢量函数的一者或两者内应用,其中受影响的代码的部分通过所需的安全级别来确定。例如,在每个可能点或几乎每个可能点处应用多个额外保护类型可以提供最大安全性;在多个点处应用单个保护或者仅在单个代码点处应用多个保护类型可以提供较低的安全级别,但是可以在代码生成和/或执行期间提供改进的性能。在一些实施例中,可以贯穿生成和绑定过程在多个点处应用断裂,因为可以存在用于断裂创建的许多机会,这归因于在状态矢量函数的步骤当中在其构造期间生成许多线性和仿射操作。

[0096] 在一些实施例中,使状态矢量函数对的一个成员比另一个成员更紧致可能是有用的。这可以例如通过使所述对的另一个成员的计算花费更大来完成。作为特定示例,当状态矢量函数对的一个成员在诸如智能卡等的暴露和/或功率受限的硬件上使用时,对于状态矢量函数对的硬件驻留的成员可以优选地比本文公开的其他实施例中更紧致。为此,可以使状态矢量函数对的对应服务器驻留或其他非暴露成员的计算的花费显著更高。作为特定示例,与如所公开的使用相对大量系数和如对于状态矢量函数生成技术所预期的不同的是,可以使用重复算法。重复算法可以使用可预测流生成过程或类似源提供的系数,所述可预测流生成过程诸如随机数生成器,其使用完全确定了所生成的序列的种子。这样的生成器的合适示例是基于 ARC4 的伪随机生成器。在一些实施例中,诸如在可用 RAM 或类似存储器相对受限的情况下,使用较小元素大小的变量可以是优选的。伪随机数生成器可以用于生成所有矩阵元素和位移矢量元素。可以应用合适的约束以确保产生的函数的可逆性。为了求逆,可以通过种子的知识来再生所生成的矩阵,代价是:在 $Z/(2^n)$ 上创建在曝露对成员中使用的完备流、相反地读取、对每个矩阵用乘法求逆,以及对位移中的每个矢量元素用加法求逆。因此,诸如智能卡的受限资源设备可以被适配为执行状态矢量函数对之一,而系

统作为整体仍然得到如本文所公开的完整状态矢量函数系统的至少一些益处。

[0097] 安全通信管道

如图 29 的框图所示，如本文所公开的基函数可以用于提供从一个或多个平台上的一个或多个应用到一个或多个其他平台上的一个或多个应用的安全通信管道(即，e- 链路)。相同的过程可以用于保护在单个平台上的从一个子程序到另一个子程序的通信。简言之，基函数对f_K, f_{K-1}可以用于通过在管道的各端执行像密码那样的加密和解密来保护该管道。在实施例中，基函数对f_K, f_{K-1}可以应用于管道开始和管道结束，并且还应用于应用及其平台，因此将它们绑定在一起并将它们绑定到管道。这保护(1)应用到管道开始，(2)管道开始到管道结束，以及(3)管道结束到应用信息流。

[0098] 实施这样的过程的说明性方式如下。首先，使用随机或伪随机过程来生成密钥 K。然后使用密钥 K 和随机化信息 R 来生成基函数对f_K, f_{K-1}。然后将基函数应用于管道开始和管道结束，使得在运行时，管道开始计算f_K并且管道结束计算f_K^{-1}。然后可以抛弃密钥 K，因为执行受保护代码不需要它。在诸如此的应用中，基函数规范将是用于f_K, f_{K-1}的基于密码的规范(类似于用于 AES 加密和解密的 FIPS-197)。隐身罩(Cloaked)基函数是光滑基函数的特定实现(上文的管道开始和管道结束)，所述光滑基函数被设计为挫败攻击者的以下尝试：找到 K、对基函数求逆(即破坏加密)，或者破坏上面示出的任何绑定。即，光滑基函数是这样的基函数，其直接实现f_K或f_K^{-1}，而未增加任何模糊。隐身罩基函数仍然计算f_K或f_K^{-1}，但是其以不直接得多的方式来这样做。其实现利用模糊熵 R 以找到用于实现f_K或f_K^{-1}的随机选择的难以跟随的技术。用于创建和使用隐身罩基函数的技术的进一步示例在本文进一步详细地提供。

[0099] 函数索引交织

为了进行保护以免受同态映射攻击，本文公开的实施例可以使用这样的函数来替代矩阵函数，所述函数是(1)宽输入的；即，包括单个输入的比特的数量大，使得可能的输入值的集合极大，以及(2)深度非线性的；即，所述函数不大可能通过 i/o 编码(即，通过单独地重编码单独输入和单独输出)被转换成线性函数。使输入宽使得通过在所有输入上对函数制表的暴力求逆消耗不切实际的大量存储器，并且深度非线性防止同态映射攻击。

[0100] 一些实施例可以使用“函数索引交织”，其可以提供深度非线性的扩散和 / 或混乱部件。如果并且只有如果从矢量到矢量的函数不能通过矩阵与任意单独输入和输出编码一起来实现的情况下，从矢量到矢量的函数才是深度非线性的。如果其不是深度非线性的，则其“直到 I/O 编码是线性的”(“直到 I/O 编码的线性”是在对白盒 AES 的 BGE 攻击中被利用的弱点)。

[0101] 函数索引交织允许要通过类似线性的手段解决的方程的一致深度非线性系统。其可以用于促进数据相关处理、一种动态多样性的形式，其中不仅计算的结果而且计算本身的性质取决于数据。图 30 示出了示例函数索引交织处理的处理流程图，该示例函数索引交织处理将单个 4x4 函数与一系列 4x4 函数进行交织。1x1 函数与 1x1 函数系列情况允许任意种类的函数的组合，诸如将密码与其自身进行组合(在 3DES 的精神内)以增加密钥空间；

将不同的密码相互组合；将标准密码与其他函数进行组合；以及将硬件和软件功能组合到单个功能中。

[0102] 在图 30 中示出的示例实现中，正方形框表示双射函数，典型地但不必然通过矩阵来实现。三角形具有与它接触的正方形框相同的输入，并且用于控制在多个右侧函数当中进行选择的开关，其中输入与输出交织左侧和右侧输入与输出，如所示出的那样：

- 如果左侧框和右侧框为一对一的，所以为整个函数；
- 如果左侧框和右侧框是双射的，所以为整个函数；
- 如果左侧框和右侧框是 MDS（最大距离可分），所以为整个函数，无论是双射与否。

[0103] 如果三角形和所有框是线性的并且被随机选择，则（通过观察）超过 80% 的构造是深度非线性的。

[0104] 在本文公开的示例实施例中，函数索引交织在f_K, f_K^{-1}规范中出现四次。每次它包括三个针对某个 4x4 矩阵 M 的 4x4 线性映射。函数索引交织的每个实例具有单个左侧函数和 2^4=16 个右侧函数。

[0105] 显著地，函数索引交织还可以是嵌套的，使得左函数或右函数系列可以自身是函数索引交织的实例。在这样的配置中，结果是函数索引交织的递归实例。一般而言，与非递归实例相比，这样的实例通常对于攻击者而言是更难理解的；即，在函数索引交织中的递归级别应当增加模糊性的级别。

[0106] 进一步的示例实施例和函数索引交织的对应数学处理在第 2.9 节中提供，并且具体地，在附录的第 2.9.2 节和图 8 中具体地提供。

[0107] 标记 I 系统

本文详细描述了三种特定示例实施例，称为标记 I、II 和 III 系统。标记 I 的示例性实现在图 31 的处理流程图中呈现。在该示例中，正方形框表示混合布尔算术（MBA）多项式编码的矩阵。MBA 多项式数据和操作编码的歧义性很可能是非常高的并且很可能随着多项式的次数快速增加。每个矩阵被独立编码，并且接口编码不需要匹配。因此，2x2 重编码不能与前驱和后继线性合并。中心构造是函数索引交织，其使得文本处理是文本相关的。使用具有移位的简单变体，在低开销的情况下，交织函数的数量可以是非常大的。例如，将 4x4 矩阵的行与列进行置换给出 576 种选择。作为另一示例，与初始和最后常量进行 XOR 给出相对非常高数量的选择。初始和最后重编码跨左固定矩阵与右可选矩阵的对应输入 / 输出将熵进行混合。在每个矩阵上的内部输入 / 输出重编码从阶$2^{3w/2}$至阶$2^{5w/2}$产生同态映射工作因数，从而允许完全“生日悖论”脆弱性——工作因数可以较高但是不可能较低。

[0108] 标记 I 系统的示例实施例和对应的数学处理在附录的第 3.5 和第 4 节以及图 4 中提供。

[0109] 然而，已经发现标记 I 类型的实现可能具有两个弱点，这两个弱点可能在某些环境中被利用：

1）静态相关性分析可以用于隔离部件。

[0110] 2）仅“开关”中的移位操作和比较是非 T 函数。所有其他部件是 T 函数，并因此使用比特切片攻击可以是递归地可分析的。

[0111] T 函数

从 w 比特字的 k 矢量到 w 比特字的 m 矢量的函数 $f:(B^w)^k \rightarrow (B^w)^m$ 映射是 T 函数，如果对于每个矢量对 $x \in (B^w)^k, y \in (B^w)^m :- y = f(x)$（$x' \neq x$ 并且 $y' = f(x')$，并且在 w 比特字中的比特编号从 0 到 $w-1$），则在 y 和 y' 不同的元素字中的最低编号比特比在 x 和 x' 不同的元素字中的最低编号比特更低。

[0112] 因此，作为 T 函数的函数将具有这样的属性，当 i>j 时，对输入元素的 2^i 比特的改变绝不会影响输出元素的 2^j 比特。典型地，字内的比特阶编号被认为是从低阶 (2^0) 到高阶 (2^{w-1}) 比特，将字视为表示二进制大小，因此这可以重新声明为：输出比特可以仅取决于相同或更低阶的输入比特。因此，或许有可能“切掉”或忽略更高比特并仍然获得有效数据。与仅使用数百个 T 函数的已知实现相对地，一些实施例还可以并入数千万个 T 函数。结果，本文公开的实施例可以更能抵抗比特切片攻击和统计攻击。

[0113] 可从在 B^w 上计算的 $\wedge, \vee, \oplus, \neg$ 与在 $Z/(2^w)$ 上的 $+, -, \times$ 一起构成使得所有操作在 w 比特字上操作的函数是 T 函数。具有 T 函数属性的模糊构造易受比特切片攻击的攻击，因为有可能通过从输入和输出矢量中的所有字下降高阶比特来从任何 T 函数中获得另一合法 T 函数。对于正确的比特移位、逐比特旋转、除法操作或者基于除数 / 模数的余数或模数操作，T 函数属性不保持，所述除数 / 模数不是 2 的幂，对于其中条件分钟做决定的函数，T 函数属性也不保持，在所述决定中，较高阶条件比特影响较低阶输出比特的值。对于条件分支和基于比较的条件执行、基于使用六种标准比较 $=, \neq, <, >, \leq, \geq$ 中的任何一个形成的条件的条件执行均可以容易地违反 T 函数条件，并且实际上，在使用基于比较的分支逻辑的正常代码中，违反 T 函数条件比服从 T 函数条件更容易。

[0114] 外部和内部脆弱性和攻击抵抗

通过重复地应用双射函数对 f_K, f_K^{-1} 中的任一个（其中 f_K, f_K^{-1} 是 T 函数），或许有可能精确地表征使用比特切片攻击的计算。在这样的攻击中，这些函数的操作被认为忽略了除低阶比特以外的所有比特，并且然后忽略低阶的两个比特，等等。这提供信息，直到达到完整字大小（例如，32 比特）为止，在达到完整字大小的点处，关于函数如何表现的完整信息可以是可获得的，其等价于密钥 K 的知识。这是外部脆弱性。当攻击获得实现细节的知识时，其这样做而不用对实现那些细节的代码的进行任何检查，并且可以如对黑盒实现的自适应已知明文攻击那样被执行。

[0115] 如果所述对的函数具有以下属性，则可以存在较不严重的脆弱性：每个函数充当在特定域上的特定 T 函数，并且独特 T 函数的数量低。在该情况中，统计分桶攻击可以表征每个 T 函数。然后，如果域可以被类似地表征而也不用对代码进行任何检查，则使用自适应已知明文攻击，攻击者可以完整表征所述对的成员的功能性、完全绕过其保护，仅使用黑盒方法。明显地，可以期望具有有效数量的独特 T 函数以挫败以上攻击。在标记 III 类型实现中，例如，每区段存在 10^8 以上个独特 T 函数，并且在整体上存在 10^{40} 以上个 T 函数。本文进一步详细地描述了标记 III 类型实现。

[0116] 在一些情况中，实现对可以包括实现全级联的函数，即，每个输出取决于每个输入，并且平均而言，改变一个输入比特会改变一半输出比特。内部脆弱性的示例可以出现在

标记 II 类型实现中，其中，通过在某点处“切割”实现，或许有可能找到对应于矩阵的子实现（部件），使得相关性级别精确地是 2x2（在该情况中，部件是混合器矩阵）或 4x4（在该情况中，其是 L、S 或 R 矩阵之一）。一旦这些已经被隔离，则线性函数的属性允许这些矩阵的非常有效的表征。这是内部攻击，因为其需要非黑盒方法：其实际上需要对实现的内部进行检查，即是静态的（以确定相关性）还是动态的（以通过基于线性度的分析来表征矩阵）。

[0117] 作为一般性规则，防止更多的外部攻击，并且迫使潜在的攻击者依赖于不断增加的细粒度的内部攻击，攻击者的工作变得越难，并且最特别地，攻击者变得越难进行自动化。自动化的攻击是尤其危险的，因为它们可以有效地提供类破裂，这允许给定技术的所有实例被可以广泛分发的工具破坏。

[0118] 因此，本文公开的实施例可以借助于可变和不断变复杂的内部结构和不断更具变化的防御来提供这样的环境，在所述环境中，实例的任何完全破裂需要许多子破裂，所需的子破裂在实例间不同，被攻击的部件的结构和数量在实例间不同，并且所采用的保护机制在实例间不同。在该情况中，使攻击自动化变为足够大的任务，从而阻碍攻击者尝试它。在相当长时间内，时间将花费来建立这样的攻击工具，所部署的保护可以已经被更新或者否则可以已经移动到新的技术，对于所述新的技术，攻击工具的算法不再够用。

[0119] 标记 II 系统

根据在图 23 和 12 中呈现的实施例的示例标记 II 类型实现的框图。图 23 呈现对在图 12 中出现 4 次的“基核函数”的处理。在图 5 和 6 中示出标记 II 类型系统的完整执行流程，并且该执行流程在附录第 5.1 节中参照图 5 和 6 被进一步详细描述。

[0120] 在根据标记 II 类型实施例的实现中，重编码的显式使用是通过 K 选择的功能性的部分。右侧的重编码和置换是从每核心 16 个配置和整体上 65,536 个配置的对中文本相关地选择的。然而，整体上 65,536 的 T 函数计数对于许多情况都可能是过低的；即使忽略内部结构并使用统计分桶的盲比特攻击也可能足以在给予足够攻击时间的情况下破裂标记 II 实现。

[0121] 标记 II 类型实现的平衡在图 12 中示出。如所示出的初始和最终置换和重编码是随机静态选择的。核心 1&2 之间和核心 3&4 之间的交换侧以及核心 2&3 之间的半交换确保跨整个文本宽度的文本相关。然而，高度常规的结构促进了通过内部相关性分析的部件隔离。一旦部件被隔离，就可以通过比特切片分析来分析 T 函数。非 T 函数部分是简单的并且可以使用直接攻击来破裂非 T 函数。因此，标记 II 实现是有效的，并且在许多应用中是有用的，但是可能向足够的访问和努力妥协。

[0122] 标记 II 提议类似于标记 I，因为其具有固定的内部结构，仅在基函数实现对当中有系数变化。在附录的第 5.1 节中提供了关于标记 II 实现的示例实施例和对应的数学处理的进一步描述。

[0123] 标记 III 系统

与上文描述的标记 I 和标记 II 实现相对地，根据本文公开的实施例的标记 III 基函数设计可以包括以下属性：

- 非常规且密钥确定的结构，使得攻击者不能预先知道结构的细节；

- 高度数据相关的功能性：改变数据会改变对数据的处理，使得统计分桶攻击是资源密集的；

- 相对极高的 T 函数计数(易受递归比特切片攻击的单独子函数的数量),使得对其 T 函数的盲比特切片攻击不可行 ;

- 冗余和隐式交叉检查数据流,使得代码修改攻击是高度资源密集的 ;以及

- 全方位引起模糊的相关性,使得基于相关性的分析是资源密集的。

[0124] 图 13 示出示例标记 III 类型实现的一部分中的执行流程的示意性表示。与关于标记 I 和标记 II 类型实现描述的示例执行流程类似地,每个部件可以表示功能、处理、算法等等,箭头表示它们之间的潜在执行路径。在不同箭头通向部件内的不同点的情况下,将理解的是,部件的不同点可以被执行,或者部件内的不同执行路径可以被选择。如图 13 中所示,标记 III 类型实现可以提供非常规、密钥相关、数据相关、数据流冗余、交叉链接、交叉检查、篡改混沌结构,在函数索引交织内包含嵌套的函数索引交织。交叉链接可以是全方位的,因为在每个交织中右侧选择取决于左侧的输入而非输出,使得在每个区段内的简单代码重排序允许从右至左的交叉连接以及从左至右的交叉连接。如图 14 中所示,非常规的极细粒度的 T 函数划分使得整体 T 函数分区攻击是无效的。

[0125] 图 15 示出了本文所公开的标记 III 类型实现的一部分的另一示例示意图。如图 15 中所示,初始和最后混合可以使用具有宽度 3 至 6 的 32 比特字的线性变换。可以使用五至七个区段,每个区段包含函数索引交织的 3 带递归实例。每个带为 3 至 6 个元素宽,全部三个带总共有 12 个元素。矩阵被 I/O 置换和 I/O 旋转,每区段给出超过 10 亿个 T 子函数 :整个基函数具有超过 10^{40} 个 T 子函数。数据流复制、随机交叉连接以及随机检查也可以与代码重排序相组合地使用,从而产生全方位交叉相关性。

[0126] 在标记 III 类型系统中可以使用的多个不同防御在图 16 中总体示出。它们包括诸如以下的特征 :

- 与断裂变换(动态数据编码识别)的存储器置乱,这隐藏数据流 ;

- 随机交叉链接、交叉俘获以及变量相关编码,这导致普遍的内相关以及混沌篡改响应 ;

- 置换多项式编码以及函数索引交织,这束缚线性攻击 ;

- 可变的随机选择的结构,这束缚预先知识攻击 ;以及

- 功能性与运行时数据高度相关,从而减小可重复性并束缚统计分桶攻击。

[0127] 在附录的第 6 节中提供了关于标记 III 类型的实现的进一步细节。在附录的第 3.3 节中提供了用于在 $Z/(2^w)$ 上创建可逆矩阵的相关过程。如所示出和描述的,也可以使用初始和 / 或最后混合 stelps,其示例在附录的第 2.8 节中提供。

[0128] 通过用将每个输入混合到每个输出中的 2x2 双射矩阵替换条件交换,我们可以精确地采取相同的网络拓扑,并且产生初始时将基函数的每个输入与每另一个进行混合的混合网络,并且我们可以最后采用另一这样的网络以将基函数的每个输出与每另一个进行混合。如上面所提到的,混合不是完全均匀的,并且可以利用由混合步骤替换的条件交换来减小其偏置。区段的输入和输出还可以被细分,例如如附录的第 6.2.3-6.2.7 节中进一步细节所描述的以及如图 11 中所示出的。

[0129] 数据流复制

一些实施例可以包括数据流复制技术。例如,如下文所描述的,对于非 JUMP……ENTER 或 EXIT 的每个指令,可以拷贝该指令使得原始指令紧跟在其拷贝后面,并且可以为所有拷

贝的指令选择新检测器，使得如果 x 和 y 是指令，其中 y 是 x 的拷贝，则：

1）如果 x 输入 ENTER 指令的输出，则对应的 y 输入使用相同的输入；

2）如果 x 输入具有拷贝 v 的原始指令 u 的输出，则对应的 y 输入从与 u 输出对应的 v 输出进行输入，x 输入从 u 输出进行输入；以及

3）如果 x 输出到 EXIT 指令，则对应的 y 输出输出到特殊的未使用宿节点，从而指示其输出被丢弃。

[0130] 因此，除了分支外，所有的计算具有原始和拷贝出现。

[0131] 为了实现该变换，我们如下继续。

[0132] 我们添加新指令 JUMPA（“任意跳转”），其是在控制流图（cfg）形式中具有*两个*目的地的*非条件*分支，正如条件分支那样，但是其没有输入：替代的是，JUMPA 在其两个目的地之间随机进行选择。JUMPA 不是 VM 指令集的实际部分，并且在f_K或f_K^{-1}的最后混淆实现中将没有 JUMPA 发生。

[0133] 我们在以下变换过程中使用 JUMPA：

1）如果实现还没有已经是 SMA（静态躲赋值）形式，则将其转换成 SMA 形式；

2)对于实现$X_1,\ldots,X_k$中的 BB 的每个$BB\ X_i$，通过以下方式来用三个 BB C_i, X_i, X'_i来替代它：创建等于 X_i 的新$BB\ X'_i$，并且添加仅包含单个以 X_i 和X'_i二者为目标的 JUMPA 指令的新BBC_i，制作 X_i 和X'_i——C_i'的 JUMPA 的两个目标，以及使指向 X_i 的每个非 JUMPA 分支目标替代地指向C_i。

[0134] 3)将实现转换成 SSA 形式(静态单赋值)，隔离每个 X_i 和X'_i中的局部数据流，但是 X_i 和X'_i中对应的指令仍然计算相同值。

[0135] 4）将每个X'_i中的所有代码合并回其 X_i 中，在合并中使来自 X_i 和X'_i的指令交替，使得对应的指令对是相继的：首先是 X_i 指令，并然后是对应的X'_i指令。

[0136] 5）使是 C_i 的每个分支目标替代地指向对应的 X_i，并且移除所有 C_i 和X'_i BB。此时，数据流已经被复制，CFG 的原始形状已经恢复，并且实现没有 JUMPA 指令。要记住哪些指令在每个 X_i 中对应以在将来使用。

[0137] 关于控制流复制的进一步细节在附录第 5.2.6 节中提供且关于图 9 进行了描述，图 9 示出根据本文公开的实施例的用于控制流复制的示例过程。

[0138] 分式和分式函数

一般地，当产生经编码的输出时，消耗精确相同的假设编码，使得经编码的操作$z=f(x,y)$变为$z'=f'(x',y')$，其中对于编码e_x, e_y, e_z，$(x', y', z')=(e_x(x), e_y(y), e_z(z))$，并且其中$f'=e_z\circ f\circ[e_x^{-1}, e_y^{-1}]$。

[0139] 在一些实施例中，以下可能是有利的：输出具有一个编码的值，并且后续输入假设的一些其他编码。如果 x 被输出为$e_1(x)$并且随后消耗假设的编码e_2，则实际上我们已经向未编码的值应用了$e_2^{-1}\circ e_1$。这样的在其中产生值的编码与假设的编码之间的故意失配被称为“分式”。如果编码是线性的，则分式函数$e_2^{-1}\circ e_1$也是，且如果它们是置换多项式，则分

式函数 $e_2^{-1} \circ e_1$ 也是。

[0140] 在一些实施例中，分式在混淆中可能是有用的，因为它们有效执行的计算在经编码的代码中不出现——执行正常联网编码的代码和借助于分式添加操作的代码的量和形式是相同的，并且看起来没有明显的方式来消除这些情况的歧义，因为编码本身趋向于是某种程度歧义的。

[0141] 注意，定义分式的属性的是分式函数，例如 $v^{-1} \circ u$。一般地，存在许多不同对消耗编码 v 和产生编码 u 的选择，它们精确地产生相同的分式函数。例如，非常有可能的是具有 $u_1, \ldots, u_k, v_1, \ldots, v_k$，使得对于 $i = 1, \ldots, k$，$v_i^{-1} \circ u_i$ 是相同的分式函数。因此，指定分式函数并不一定指定产生和消耗暗示其的编码。

[0142] 经由海量数据编码的数据加扰

在美国专利 No.7,350,085 中描述了海量数据编码(MDE)，该专利的内容通过引用并入本文。简言之，MDE 以像哈希这样的方式对存储器位置加扰，从而在每个存储上对存储器单元重编码，并通过后台处理来动态对存储器单元重编码和重定位。通过使取得和存储重编码失配，取得或存储可以执行加法或乘法同时继续看起来像简单的取得或存储。这使得攻击者难以消除仅仅混淆与有用工作之间的歧义。

[0143] MDE 被编译而非仅仅被解释，从而支持的数据结构是部分隐式的并因此是良好模糊的。实际地址时钟通过后台活动来加扰和重加扰。如图 17 中所示，访问虚拟 MDE 存储器的代码初始被写入为像其正访问普通存储器段一样。然后，通过美国专利 7,350,085 中描述方法来修改代码以采用映射技术，该映射技术对存储器中的数据和位置二者进行编码。因此，在正运行代码的底部下，被访问的位置随着时间而四处移动，并且应用于数据的编码同样地随着时间而改变。该保护技术具有相当大的开销，但是其的高度动态性质使得攻击者洞察使用其的软件的意义是艰难的。单元在被存储时被重编码并且通过后台活动被周期性地重编码。对存储的重编码与对取得的对应重编码的失配可以做转换加法或乘法(密钥可控的)。取得的项目被重编码，但是不是平滑的(即不是未编码的)。所存储的项目在存储之前不是平滑的，并且对存储重编码为动态选择的新单元编码。在没有访问所存储的数据的代码的情况下，所存储的数据是无意义的。一个程序可以具有任意数量的独特、非重叠 MDE 存储器。MDE 存储器可以作为块从一个位置移动到另一个位置，或者可以经由传输介质从一个程序传输到另一个程序。即，足够大块的消息可以以 MDE 存储器形式被传输。

[0144] 存储器的初始状态未被黑客可见活动产生，并且因此隐匿其内容是如何得到的。即，初始状态是尤其模糊的。

[0145] 经由控制流来控制混乱

在美国专利 No.6,779,114 中描述了控制流编码(CPE)，其内容通过引用并入本文。CFE 将代码片段组合到具有通过以下控制的功能性的多功能团中：寄存器切换：功能性到代码位置的多对多映射；外部熵可用的情况下高度不可重复的执行：相同的原始代码变成 CFE 代码中的许多替代执行。通过修改寄存器切换和分派代码，密钥信息可以控制什么被执行并因此控制本发明的实施例执行的计算。

[0146] 图 18 的控制流图表示的代码(其中字母表示代码片段)可以如图 19 中所示那样被编码。受保护的控制流编码示出通过组合在分派器控制下执行的段与寄存器切换选择的

“活动”段而创建的团。

[0147] CFE 被编译而不仅仅被解释，从而支持的数据结构是部分隐式的并因此是良好模糊的。团组合多个段：即，它们具有多个可能的功能性。当执行团时，其的是活动的一个或多个段通过其经由指向实际数据而非哑数据的寄存器的操作来确定。相同的段可以在多个团中出现，具有不同的数据编码：从功能性到代码位置的映射是多对多的。

[0148] 分派器可以布置为选择体现后台处理的段，从而使得难以区分后台和前台活动。可用熵用于确定执行一系列段的哪种可替代方式被采用，从而提供动态执行多样性（非重复执行）。此外，密钥信息可以用于影响分派，并因此改变所代表的算法。

[0149] 动态数据识别编码

如图 20 中所示，可以通过使用 Chaitin 的图着色分配算法来使 M 寄存器的重用最大化，仅在需要的情况下才分配单独的 M 寄存器。结果，M 寄存器被频繁重用，使得攻击者难以跟随数据流。

[0150] 为了这样做，首先可以选择模数 M、在模 M 环上的置换多项式 p、从输入产生 z 的基于输入的$1\times n$矢量矩阵 A，以及一系列常量$c_i = p(z+i)$，其中$1 \le i \le M$，其中c_i值是独特的，因为 p 是模 M 置换多项式。在大小为 M 的阵列 X 中将位置$c_1, \ldots, c_n$ $(n \le M)$作为“M 寄存器”来对待。

[0151] 在计算期间，数据可以随机移进和移出 M 寄存器，并且从 M 寄存器移动到 M 寄存器，每次移动时改变编码。一些实施例还可以随机地使得编码形成未被破坏的序列，或者可以如本文所公开的那样注入分式，其中编码不相匹配。

[0152] 给定具有在编码 e1 中的数据的分式，输入被假设为在编码 e2 中，因此计算分式函数$e3 = e2\text{-}1 \circ e1$。如果 e1、e2 是线性的，则 e3 也是。如果 e1、e2 是置换多项式，则 e3 也是。代码具有相同的形式，而无论分式存在与否；即，其是歧义的，无论分式存在与否。因此，如之前所描述的，分式可以提供注入隐藏的计算的手段，使得代码在其被添加之前和之后看起来非常相同。

[0153] 在附录第 7.8.14 节提供了动态数据识别编码的使用的附加细节和数学处理。

[0154] 交叉链接和交叉俘获

交叉链接和交叉俘获的大量应用可以提供对篡改和扰乱攻击的攻击性混沌响应，具有强得多的代码转换和海量静态分析抵抗。在实施例中，交叉链接和交叉俘获可以如下实行，如图 21 中所示的：

1）拷贝计算至少一次；

2）在原件和拷贝之间随机地交换连接。因为它们是复制件，所以结果将不会改变；

3）对所有因而产生的计算进行编码，使得复制件是独立编码的；

4）随机地取复制结果，并且注入计算：将它们的差异相加（=0），或者将一个结果与另一个的环逆（ring inverse）相乘（=1）并然后加上 0 或乘以 1（以经编码的形式）。注入的经编码的 0 加法或 1 乘法没有功能上的影响，除非篡改发生，在该情况下，代码混沌地表现。

[0155] 增加的益处是静态相关图变得比原始程序的静态相关图密集得多，使得静态分析攻击是很难的。因此，有效的篡改需要（不同编码的）复制件被正确标识并且以在不同编码下有效的相同方式来改变正确的复制件。与没有交叉链接和交叉俘获的普通篡改相比，这的实现要难得多。

[0156] 数据流复制的示例实现在附录的第 5.2.8-5.2.10 节中提供，并且在图 10 中示出。除了其在进入和退出基函数内的正常使用之外，数据流复制和交叉检查或俘获也可以使用决策块内针对数据流的这些变换来执行，包括信息从进入基函数的输出到决策块的输入的转移，以及信息从决策块的输出到退出基函数的输入的转移。

[0157] 上下文相关编码

在一些实施例中，其中实现基函数对的上下文可以是基函数的操作的组成部分。上下文包括来自应用、硬件和 / 或通信的信息。一个基函数部件的上下文还可以包括来自其他部件的信息，这些其他部件是其驻留于其中的应用的部分。

[0158] 参照图 22，基函数对的实现或类似构造可以托管在平台上，硬件或其他平台签名常量可以从所述平台得到，并且可以取决于所述平台来做出实现。对于实现而言可以优选的是驻留在包含应用中，应用签名或其他应用常量可以从所述包含应用得到，并且可以取决于所述包含应用来做出实现。

[0159] 实现还可以取这样的输入，进一步的常量签名信息可以从所述输入得到，并且可以取决于所述输入来做出实现。

[0160] 经由排序网络的偏置置换

置换可以提供偏置来将极大数量的替代存储在有限空间中。例如，行 / 列置换可以用于将非重复 4x4 矩阵变为 576 个非重复 4x4 矩阵。在一些实施例中，计算的顺序可以被置换，可以生成对运行时数据的计算的深度相关，等等。

[0161] 参照图 7，一些实施例可以首先在每个交叉链接处进行排序、比较并在大于时进行交换。为了进行置换，以 1/2 概率执行交换。容易示出 ：如果网络用比较 - 交换正确地进行排序，则其利用随机交换来进行置换，其中全范围的置换作为可能的输出。一些实施例可以使用推荐的 1/2 概率的布尔生成子来比较两个基于文本的全范围置换多项式编码的值。

[0162] 这样的排序网络以偏置方式进行置换，即，一些置换比另一些更加可能，因为交换配置的数量为 $2^{\text{阶段的数量}}$。然而，置换的计数等于要置换的元素的数量，该数量不整除交换配置的数量。尽管有偏置输出，但是优点是简单性和与非 T 功能性的高度相关性。

[0163] 经由样本选择的非偏置置换

在一些实施例中，也可以通过以下方式来生成非偏置置换 ：在元素当中取 r_1 模 n 元素来选择第一个元素（零原点），从剩余元素中随机取 r_2 模（n-1）元素来选择第二个元素，等等。以该过程，每个 r_i 是全范围基于文本的置换值。这可以几乎完美地提供无偏置和非 T 函数。然而，与针对基于排序网络的置换相比，操作可能更能隐藏在普通代码中或与普通代码交织。

[0164] 束缚比特切片分析

如上文所解释的，比特切片攻击是常用的攻击工具 ：重复地执行函数并忽略除了最低位比特之外的所有比特，并然后是最低位的两个比特、三个最低位比特等等。这允许攻击者获得信息，直到达到全部字大小（即 32 比特），在该点处，关于函数如何表现的完整信息已经被获得。

[0165] 使用 T 函数和非 T 函数部件构造的函数具有子域，在所述子域上，其是嵌入在整个域中的 T 函数，在整个域中该函数不是 T 函数。在一些实施例中，使多个这样的子域非常大可能是有利的（例如，在如本文所描述的标记 III 类型系统中，可以存在 10^{40} 以上的这样的

子域)，从而使得对这些子域的分桶攻击是高度子域密集的。在一些实施例中，自由使用也可以在其他点处由非 T 函数计算制成，诸如在决策点处、在置换中、在记录中等等。

[0166] 示例一般数据掺合机制

图 24 示出了如上文描述的海量数据编码或动态数据识别编码的典型使用的图形表示。如果至基函数的输入由这样的模糊存储器阵列、这两种技术任一种提供并且结果也通过来自模糊存储器阵列的应用获得，则攻击者变得难以分析进入或离开基函数的信息的数据流，使得对基函数的工具是更加艰难的。

[0167] 安全性刷新速率

为了有效的应用安全生命周期管理，应用典型地必须能够在正在进行中的基础上对抗攻击。作为该对抗的部分，这样的应用可以被配置为响应于包含安全更新信息的安全性刷新消息而自升级。这样的升级可以涉及补丁文件、表替换、新密码密钥以及其他安全相关信息。

[0168] 可行的安全级别是这样的安全级别，其中应用安全性被足够频繁地刷新，使得损害实例的安全性花费的时间比使损害无效的安全性刷新的时间更长，即，实例比它们可能典型地被破坏更快地被刷新。这当然是以非常高的安全性刷新速率可实现的。然而，这样的频繁刷新动作消耗带宽，并且随着我们提升刷新速率，分配给安全性刷新消息的带宽部分增加，并且可用的非安全性有效载荷带宽降低。

[0169] 明显地，于是，设计合适的安全性刷新速率对于每种应用都是需要的，因为能容忍的开销取决于上下文有很大的变化。例如，如果我们预期在云应用中仅有灰盒攻击(相邻侧信道攻击)，则与我们预期有白盒攻击(由恶意的云提供商职员进行的内部攻击)的情况下相比，我们将使用更低的刷新速率。

[0170] 对具有混沌失败的相等性的认证

假设我们具有其中认证像密码那样的应用：在提供的值 G 匹配参考值Γ(即当$G=\Gamma$)的情况下，认证成功。进一步假设我们关注当$G=\Gamma$时会发生什么，而如果不相等，我们仅仅认为无论如何认证授权不再可行。即，当$G=\Gamma$时我们成功，而如果$G\neq\Gamma$，则进一步计算可以简单地失败。

[0171] 通过向两侧应用任何无损函数，对相等性的认证不受影响：对于任何双射ϕ，我们可以等同地测试是否有$\phi(\phi)=\phi(\Gamma)$。即使ϕ是有损的，如果仔细选择ϕ使得当$G\neq\Gamma$时$\phi(G)=\phi(\Gamma)$的可能性足够低，对相等性的认证也可以以高概率保持有效(例如，如在 Unix 口令认证中那样)。基于本文之前描述的技术，我们可以简单地执行这样的测试。我们之前描述了一种通过以下方式来挫败篡改的方法：复制数据流、随机地交叉连接复制实例之间的数据流，以及执行经编码的检查以确保相等性未被损害。我们可以改编该方法来测试是否有$G=\Gamma$，或者在编码形式中，是否有$\phi(G)=\phi(\Gamma)$。

[0172] 我们注意到产生$\phi(G)$的数据流已经沿着其中$G=\Gamma$的成功路径复制了产生$\phi(\Gamma)$的数据流。我们因此针对该比较省略数据流复制步骤。然后，我们简单地进行如上文描述的交叉连接并插入检查。通过将这些计算用作将来经编码的计算的系数，我们确保如果$\phi(G)=\phi(\Gamma)$，则所有都将正常进行，而如果$\phi(G)\neq\phi(\Gamma)$，则在将继续进一步的比较的同时，

结果将是混沌的，并且其功能性将失效。此外，因为ϕ是函数，所以如果$\phi(G)\neq\phi(\Gamma)$，则我们可以肯定$G\neq\Gamma$。

[0173] 变量相关编码

在并入了利用一个或多个变量的操作的一些实施例中，变量相关编码可以用于进一步模糊相关代码的操作，其中所述一个或多个变量在它们在操作中使用期间不需要具有特定值。这样做的一种方式是使用由附近的其他操作或相关代码段使用或生成的值。因此，这样的值可以在代码区内被重复地用于不同的目的，这可以使攻击者更难区分任何单独使用或提取关于特定操作的信息，所述特定操作是关于那些值执行的。例如，如果值 x 被编码为$aX+b$，则在用于常量 a 和 b 的特定值中可以存在大量余地。在该示例中，如果在执行代码内存在可用的在 x 的生命内保持恒定的值，则它们可以用作常量 a 和 / 或 b 中的一个或多个。

[0174] 此外，对于单个定义的操作，不同的值可以在每次执行操作期间使用，使得所使用的特定值可以每次执行操作时改变。这可以充当对潜在攻击者的额外屏障，所述潜在攻击者可能不能跟踪从一个执行到另一个执行的值，如对于其他类型的清楚、加密的或混淆的代码可以预期的。继续上面的示例，第一操作$f(Y)$可以返回值 a 和 b，并且第二操作$g(Z)$可以返回值 c 和 d，每个值在一段时间内存储在存储器中。在 a 和 b 存储在存储器的时间期间可以将变量 x 编码为$aX+b$，并且在 c 和 d 存储在存储器中的时间期间可以将变量 x 编码为$cX+d$。因此，合适的变量将是经由存储器而可用的，以允许对 x 进行解码或者以合适编码来另外的操纵 x。这些值可以在所述时间之后被重写或丢弃，因为对常量进行编码仅需要在 x 由执行程序内的操作使用的时间期间是可用的。

[0175] 类似地，除了所提供的有限编码示例或作为替代，在代码执行期间生成的变量值可以用于其他目的。例如，变量值可以用于从列表或索引选择随机项作为用于伪随机数生成子的种子、作为加法、乘法或其他比例因子，等等。更一般地，在比生成的变量值预期为可用更长的持续时间内，执行代码的一部分生成的变量值可以用在其中在执行代码的另一部分处需要常量值的任何位置中。

[0176] 示例优点

本文描述的发明的实施例可以用于提供以下，其中“足够的时间段”可以基于安全生命周期管理的需要来选择，或者否则由安全生命周期管理的需要来确定：

1）黑盒安全性：作为抵抗攻击的加密钥的黑盒密码的安全性取决于足够时间段内的自适应的已知明文；

2）安全边界：在足够时间段内以编码形式将信息至 / 从周围代码安全地传入和传出；

3）密钥隐藏：防止密钥提取在足够时间段内实现；

4）保护最弱路径：即使在最弱数据路径上也在足够时间段内密码地进行保护；

5）防分区：在足够时间段内将实现分区为其构造块；

6）应用锁定：不能在足够时间段内从其保护应用提取实现；以及

7）节点锁定：不能在足够时间段内从其主机平台提取实现。

[0177] 一般地，本文公开的实施例涉及使用所公开的各种技术和系统来进行基函数编码。特定实施例在本文中（诸如在附录中）也可以被称为“透明盒”（ClearBox）实现。

[0178] 本文所公开的各种技术可以使用在本质上与由所公开的技术保护的应用中使用的那些类似的操作，如之前所描述的。即，诸如基函数、分式、动态数据识别编码、交叉链接和变量相关编码的保护技术可以使用与原始应用代码使用的那些类似的操作，使得潜在攻击者可能难以或者不可能在原始应用代码与本文公开的保护措施之间进行区分。作为特定示例，与例如已知的加密技术采用的独特函数相对地，可以使用与原始应用代码执行的操作相同或计算上类似的操作来构造，所述基函数与原始应用代码集成在一起。这样的难以或不可能区分的操作和技术在本文可以描述为“计算上类似的”。

[0179] 方法总地被设想为导致期望的结果的自相容步骤序列。这些步骤需要对物理量的物理操纵。通常，尽管不是一定的，这些量采取能够被存储、传送、组合、比较以及以其他方式操纵的电或磁信号的形式。有时主要出于通用的原因，将这些信号称为比特、值、参数、项目、元素、对象、符号、字符、项、数字等等是方便的。然而，应当注意的是，所有这些项和类似项应与合适的物理量相关联，并且仅仅是应用于这些量的方便标签。已经出于说明的目的给出了对本发明的描述，但是该描述并不意图是穷尽的或者限于所公开的实施例。许多修改和变型对于本领域的普通技术人员将是清楚的。选择这些实施例来解释本发明的原理及其实际应用，并使本领域其他普通技术人员能够理解本发明，以便实现具有可能适于其他预期使用的各种修改的各种实施例。

[0180] 本文公开的实施例可以在各种计算机系统和架构中实现并且与各种计算机系统和架构一起使用。图 32 是适于实现本文公开的实施例的示例计算机系统 3200。计算机 3200 可以包括通信总线 3201，其互连系统的主要部件，诸如中央处理器 3210；固定储存器 3240，诸如硬驱动器、闪速储存器、SAN 设备等等；存储器 3220；输入 / 输出模块 3230，诸如经由显示适配器连接的显示屏，和 / 或一个或多个控制器与相关联的用户输入设备，诸如键盘、鼠标等等；以及网络接口 3250，诸如因特网或类似接口，用于允许与一个或多个其他计算机系统的通信。

[0181] 如本领域技术人员将容易理解的，总线 3201 允许中央处理器 3210 与其他部件之间的数据通信。与计算机 3200 驻留在一起的应用一般可以存储在诸如储存器 3240 或其他本地或远程存储设备的计算机可读介质上或经由该计算机可读介质来访问。一般地，所示出的每个模块可以与计算机整合，或者可以是分离的并通过其他接口来访问。例如，存储器 3240 可以是诸如硬驱动器的本地储存器，或者是诸如网络附着的存储设备的远程储存器。

[0182] 许多其他设备或部件可以以类似方式连接。相反地，对于本文公开的实际实施例，示出的所有部件不需要都存在。这些部件可以以与示出的不同的方式来互连。诸如所示的那些的计算机的操作是本领域中容易知道的并且在本申请中未详细讨论。用于实现本公开的实施例的代码可以存储在计算机可读存储介质中，诸如存储器 3220、储存器 3240 或它们的组合中的一个或多个。

[0183] 更一般地，本文公开的各种实施例可以包括计算机实现的处理的形式和用于实施那些处理的装置或者以所述形式和装置来体现。实施例还可以以计算机程序产品的形式来体现，所述计算机程序产品具有包含体现在非暂态和 / 或有形介质中的指令的计算机程序代码，所述介质诸如软盘、CD-ROM、硬驱动器、USB（通用串行总线）驱动器，或任何其他机器可读存储介质。当这样的计算机程序代码被加载到计算机中或者由计算机执行时，计算机可以成为用于实施本文公开的实施例的装置。例如，实施例还可以以计算机程序代码的形

式来体现，而无论是存储在存储介质中、加载到计算机中和／或由计算机执行还是通过一些传送介质被传输，诸如通过电布线或线缆、通过光纤或经由电磁辐射来传输，其中当计算机程序代码被加载到计算机中并由计算机执行时，计算机成为用于实施本文公开的实施例的装置。当实现在通用处理器上时，计算机程序代码可以配置处理器以创建专用逻辑电路。在一些配置中，存储在计算机可读存储介质上的计算机可读指令集可以通过通用处理器来实现，所述计算机可读指令集可以将通用处理器或包含通用处理器的设备变换成配置为实现或执行这些指令的专用设备。实施例可以使用硬件来实现，所述硬件可以包括处理器，诸如以硬件和／或估计来体现根据所公开的主题的实施例的技术的全部或部分的通用微处理器和／或专用集成电路(ASIC)。

[0184] 在一些实施例中，本文公开的各种特征和功能可以通过计算机系统内的和／或计算机系统执行的软件内的一个或多个模块来实现。例如，根据本文公开的一些实施例的计算机系统可以包括配置为进行以下的一个或多个模块：接收现有计算机可执行代码、修改如本文公开的代码，以及输出修改后的代码。每个模块可以包括一个或多个子模块，诸如其中配置为修改现有计算机可读代码的模块包括用于生成基函数、将基函数与代码进行掺合以及输出掺合后的代码的一个或多个模块。类似地，其他模块可以用于实现本文公开的其他功能。每个模块可以配置为执行单个功能，或者一个模块可以执行多个功能。类似地，每个功能可以通过单独或协作操作的一个或多个模块来实现。

[0185] 已经通过示例描述了一个或多个当前优选的实施例。对于本领域技术人员而言将清楚的是，可以做出多个变型和修改而不偏离如权利要求中所限定的本发明的范围。

[0186] 1. 介绍

本文档解决以下问题：创建纲领性实现 F、G 的对，分别对于双射函数f, f^{-1}的对，使得：

(1) 给定对 F 和值 y 的白盒访问，“难以”找到 x 使$y = f(x)$；

(2) 给定对 G 和值 x 的白盒访问，“难以”找到 y 使$x = f^{-1}(y)$；

(3) 给定对 F 的白盒访问，“难以”找到对于f^{-1}的实现 Q；以及

(4) 给定对 G 的白盒访问，“难以”找到对于f的实现 P。

[0187] 我们注意到，信息 K 足以容易确定f, f^{-1}可以视为用于对称密码的密钥，其中 F 和 G 根据密钥 K 来加密和解密。

[0188] 我们未指定我们关于“难以”是和意义。最低限度，我们希望与解决上述问题(1)-(4)中的任何一个相比，选择 K 和生成 F、G 是需要显著更少的努力的。

[0189] 附录

记号	含义
B	比特集合 ={0,1}
N	自然数集合 ={1,2,3……}
N_0	有限素数集合 ={0,1,2……}
Z	整数集合 ={……,-1,0,1……}
$x :- y$	x 使得 y

$x \leftarrow y$	将 x 设置为 y
$x \textbf{ iff } y$	如果且仅如果 y，x
[A]	如果断言 A 为真，则 1；否则 0
$x \| y$	元组或矢量 x 和 y 的拼接
$x \wedge y$	x 和 y 的逻辑或按比特与
$x \vee y$	x 和 y 的逻辑或按比特包含
$x \oplus y$	x 和 y 的逻辑或按比特排他
$\neg x$ 或 $\bar{x}$	x 的逻辑或按比特非
x^{-1}	x 的倒数
$\lceil x \rceil$	使 $x \leq k$ 的最小整数 k
$\lfloor x \rfloor$	使 $k \leq x$ 的最大整数 k
$f\{S\}$	在 MF f 下的集合 S 的像
$f(x) = y$	向 x 应用 MF f 产生且仅产生 y
$f(x) \rightarrow y$	向 x 应用 MF f 可以产生 y
$f(x) \nrightarrow y$	向 x 应用 MF f 不能产生 y
$f(x) = \perp$	向 x 应用 MF f 的结果未被定义
M^{T}	矩阵 M 的转置
$\|S\|$	集合 S 的素数
$\|V\|$	元组或矢量 V 的长度
$\|n\|$	数 n 的绝对值
$(x_1, \ldots, x_k)$	具有元素 $x_1, \ldots, x_k$ 的 k 元组或 k- 矢量
$\{m_1, \ldots, m_k\}$	MF $m_1, \ldots, m_k$ 的 k- 聚合
$(m_1, \ldots, m_k)$	MF $m_1, \ldots, m_k$ 的 k- 凝聚
$\{x_1, \ldots, x_k\}$	$x_1, \ldots, x_k$ 的集合
$\{x \mid C\}$	使得 C 的 x 的集合
$\{x \in S \mid C\}$	使得 C 的集合 S 的成员 x 的集合
$\Delta(x, y)$	从 x 到 y 的汉明距离（= 改变的元素位置的数量）
$S_1 \times \cdots \times S_k$	集合 $S_1, \ldots, S_k$ 的笛卡尔积
$m_1 \circ \cdots \circ m_k$	MF $m_1, \ldots, m_k$ 的复合
$x \in S$	x 是集合 S 的成员

$S \subseteq T$	集合 S 包含在集合 T 中或者等于集合 T
$S \subsetneq T$	集合 S 真包含在集合 T 中
$\sum_{i=1}^{k} x_i$	$x_1, \ldots, x_k$ 的和
GF(n)	具有 n 个元素的伽罗瓦域(= 有限域)
Z/(k)	整数模 k 的有限环
id_S	集合 S 上的恒等函数
rand(n)	$\{0, 1, \ldots, n-1\}$ 上的均匀随机变量
$\mathbf{extract}[a,b](x)$	比特串 x 的位置 a 至 b 中的比特字段
$\mathbf{extract}[a,b](v)$	$(\mathbf{extract}[a,b](v_1), \ldots, \mathbf{extract}[a,b](v_k))$，其中 $v = (v_1, \ldots, v_k)$
$\mathbf{interleave}(u,v)$	$(u_1 \| v_1, \ldots, u_k \| v_k)$，其中 $u = (u_1, \ldots, u_k)$ 并且 $v = (v_1, \ldots, v_k)$

表 1 记号

缩写	展开
AES	高级加密标准
agg	聚合
API	应用程序接口
BA	布尔算术
BB	基块
CFC	控制流图
DES	数据加密标准
DG	有向图
dll	动态链接库
GF	伽罗瓦域(= 有限域)
IA	干预聚合
iff	如果且仅如果
MBA	混合布尔算术
MDS	最大距离可分
MF	多函数
OE	输出扩展
PE	部分评估
PLPB	逐点线性分区双射
RSA	Rivest Shamir Adleman
RNS	余数系统
RPE	相反部分评估

TR	抗篡改
SB	替换盒
SBE	基于软件的实体
so	共享对象
VHDI	甚高速集成电路硬件描述语言

表 2 缩写。

[0190] 2. 术语和记号

我们写“:-”来表示“使得”,并且我们写“iff”来表示“如果且仅如果”。表 1 概述了本文采用的许多记号,并且表 2 概述了本文采用的许多缩写。

[0191] 2.1 集合、元组、关系以及函数。对于集合 S,我们写 |S| 来表示 S 的素数(即,集合 S 中的成员的数量)。我们还使用 |n| 来表示数 n 的绝对值。

[0192] 我们写$\{m_1, m_2, \ldots, m_k\}$来表示其成员是$m_1, m_2, \ldots, m_k$的集合。(因此,如果$m_1, m_2, \ldots, m_k$均不同,则$|\{m_1, m_2, \ldots, m_k\}| = k$。)我们还写$\{x \mid C\}$来表示形式 x 的使得条件 C 保持的所有条目的集合,其中 C 正常情况下是取决于 x 的条件。

[0193] 2.1.1 笛卡尔积、元组以及矢量。在 A 和 B 是集合的情况下,AxB 是 A 和 B 的笛卡尔积 ;即,所有这样的对 (a,b) 的集合,其中$a \in A$(即,a 是 A 的成员)以及$b \in B$(即,b 是 B 的成员)。因此,我们具有$(a, b) \in A \times B$。一般而言,对于集合$S_1, S_2, \ldots, S_k$,$S_1 \times S_2 \times \cdots \times S_k$的成员是形式为$(s_1, s_2, \ldots, s_k)$的 k 元组,其中对于$i = 1, 2, \ldots, k$,$s_i \in S_i$。如果$t = (s_1, \ldots, s_k)$是元组,则我们写$|t|$来表示 t 的长度(在该情况中,$|t| = k$;即,元组具有 k 个元素位置)。对于任何 x,我们认为 x 与 (x)——长度为 1 的其唯一的元素为 x 的元组——相同。如果元组的所有元素属于相同集合,则我们称其为该集合上的矢量。

[0194] 如果 u 和 v 是两个元组,则 u||v 是它们的拼接 :通过创建按顺序包含 u 的元素并然后按顺序包含 v 的元素的元组来获得 |u|+|v| 长度的元组 :例如$(a, b, c, d) \| (x, y, z) = (a, b, c, d, x, y, z)$。

[0195] 我们认为括号在笛卡尔积中是有意义的 :对于集合 A、B、C,$(A \times B) \times C$的成员看起来像$((a, b), c)$,而$A \times (B \times C)$的成员看起来像$(a, (b, c))$,其中$a \in A, b \in B$并且$c \in C$。类似地,$A \times (B \times B) \times C$的成员看起来像$(a, (b_1, b_2), c)$,其中$a \in A, b_1, b_2 \in B$并且$c \in C$。

[0196] 2.2.1 关系、多函数(MF)以及函数。k 个集合(其中我们必须具有$k \geq 2$)的笛卡尔积$S_1 \times \cdots \times S_k$上的 k 元关系是任何集合$R \subseteq S_1 \times \cdots \times S_k$。通常,我们将对二元关系感兴趣 ;即,对于(不一定不同的)两个集合 A、B 的关系$R \subseteq A \times B$。对于这样的二元关系,我们写$a\ R\ b$来指示$(a, b) \in R$。例如,在 R 是实数的集合的情况下,在实数对上的二元关系$< \subseteq \mathbf{R} \times \mathbf{R}$是使得 x 小于 y 的所有实数对 (x,y) 的集合,并且当我们写$x < y$时,

意思是$(x,y)\in <$。

[0197] 记号$R::A\mapsto B$指示$R\subseteq A\times B$，即 R 是 AxB 上的二元关系。该记号类似于用于下面的函数的记号。其意图是指示二元关系被解释为多函数(MF)，计算的关系抽象——不一定是确定性的——其从集合 A 取输入并返回在集合 B 中的输出。在函数的情况下，该计算必须是确定性的，而在 MF 的情况下，该计算不需要是确定性的，并因此其对于其中外部事件可能影响给定过程内的执行进展的许多软件而言是更好的数学模型。A 是 MF R 的域，并且 B 是 MF R 的值域。对于任何集合$X\subseteq A$，我们定义$R\{X\}=\{y\in B\mid \exists x\in X:-(x,y)\in R\}$。$R\{X\}$是 X 在 R 下的像。对于 MF $R::A\mapsto B$并且$a\in A$，我们写$R(a)=b$来意指$R\{\{a\}\}=\{b\}$，我们写$R(a)\to b$来意指$b\in R\{\{a\}\}$，我们写$R(a)\nrightarrow b$来意指$b\notin R\{\{a\}\}$，并且我们写$R(a)=\perp$(读“R(a)是未定义的”来意指不存在$b\in B:-(a,b)\in R$。

[0198] 对于二元关系$R::A\mapsto B$，我们定义$R^{-1}=\{(b,a)\mid (a,b)\in R\}$。$R^{-1}$是 R 的倒数。

[0199] 对于二元关系$R::A\mapsto B$和$S::B\mapsto C$，我们通过$S\circ R=\{(a,c)\mid \exists b\in B:-a\ R\ b\text{ and }b\ S\ c\}$来定义$S\circ R::A\mapsto C$。$S\circ R$是 S 与 R 的复合。二元关系的复合是联合；即，对于二元关系Q,R,S，$(S\circ R)\circ Q=S\circ(R\circ Q)$。因此，对于二元关系$R_1,R_2,\ldots,R_k$，我们可以自由地写$R_k\circ\cdots\circ R_2\circ R_1$而没有括号，因为该表达无论我们将它们放在何处都具有相同的意义。注意$(R_k\circ\cdots\circ R_2\circ R_1)\{X\}=R_k\{\cdots\{R_2\{R_1\{X\}\}\}\cdots\}$，其中我们首先取 X 在$R_1$下的像，并然后取该像在$R_2$下的像，以此类推直到倒数第二个像在$R_k$下的像，这是复合中左边的$R_i$以取像操作的相反顺序来写的原因，正如$R_i$在取像表达中右边那样。

[0200] 在对于$i=1,\ldots,k$有$R_i::A_i\mapsto B_i$的情况下，$R=[R_1,\ldots,R_k]$是二元关系$:-R::A_1\times\cdots\times A_k\mapsto B_1\times\cdots\times B_k$，并且$R(x_1,\ldots,x_k)\to(y_1,\ldots,y_k)$ **iff** $R_i(x_i)\to y_i\text{ for }i=1,\ldots,k$。$[R_1,\ldots,R_k]$是$R_1,\ldots,R_k$的聚合。

[0201] 在对于$i=1,\ldots,n$有$R_i::A_1\times\cdots\times A_m\mapsto B_i$的情况下，$R=\langle R_1,\ldots,R_n\rangle$是二元关系:-

$$R::A_1\times\cdots\times A_m\mapsto B_1\times\cdots\times B_n$$

并且

$$R(x_1,\ldots,x_m)\to(y_1,\ldots,y_n)\quad\textbf{iff}\quad R_i(x_1,\ldots,x_m)\to y_i\text{ for }i=1,\ldots,n$$

。$\langle R_1,\ldots,R_k\rangle$是$R_1,\ldots,R_k$的凝聚。

[0202] 我们写 $f: A \mapsto B$ 来指示 f 是从 A 到 B 的函数，即，$f: A \mapsto B :=$，对于任何 $a \in A$ 且 $b \in B$，如果 $f(a) \to b$，则 $f(a) = b$。对于任何集合 S，$\mathbf{id}_S$ 是对于所有 $x \in S$ 有 $\mathbf{id}_S(x) = x$ 的函数。

[0203] 2.1.3 有向图、控制流图以及支配者(dominator)。有向图(DG)是有序对 $G = (N, A)$，其中集合 N 是节点集合并且二元关系 $A \subseteq N \times N$ 是弧关系或边关系。$(x, y) \in A$ 是 G 的弧或边。

[0204] DG $G = (N, A)$ 中的路径是节点序列 $(n_1, \ldots, n_k)$，其中对于 $i = 1, \ldots, k$，$n_i \in N$，并且对于 $i = 1, \ldots, k-1$，$(n_i, n_{i+1}) \in A$。$k - 1 \geq 0$ 是路径的长度。最短可能路径具有长度为零的形式 (n_1)。路径 $(n_1, \ldots, n_k)$ 是无环的，如果且仅如果没有节点在其中出现两次；即，如果且仅如果不存在使 $n_i = n_j$ 的 $1 \leq i < j \leq k$ 的索引 i、j。对于集合 S，我们定义 $S^r = S \times \cdots \times S$，其中 S 出现 r 次并且 X 出现 r-1 次(使得 $S^1 = S$)，并我们定义 $S^+ = S^1 \cup S^2 \cup S^3 \cup \ldots$——对于所有可能长度的 S 的所有笛卡尔积的无限并集。然后，G 中的每个路径是 N^+ 的元素。

[0205] 在有向图 (DG) $G = (N, A)$ 中，如果在 G 中存在从 x 开始并在 y 结束的路径，则节点 $y \in N$ 是是从节点 $x \in N$ 可达的。(因此，每个节点都是从其自己可达的。)$x \in N$ 的到达是 $\{ y \in N \mid$ y 是从 x 可达的 $\}$。如果且仅如果两个以下条件之一递归地保持，则两个节点 x、y 在 G 中相连：

(1)存在 G 的其中 x 和 y 二者都出现的路径，或者

(2)G 中存在节点 $z \in N$，使得 x 和 z 相连并且 y 和 z 相连。

[0206] (如果 x=y，则单元素集合(即，长度为 1)路径 (x) 是从 x 到 y 的路径，所以 G 的每个节点 $n \in N$ 连接到其自己。)如果且仅如果 G 的每个节点对 $x, y \in N$ 都相连，DG $G = (N, A)$ 是连通图。

[0207] 对于每个节点 $x \in N$，$|\{y \mid (x, y) \in A\}|$，A 中在 x 处开始并在某个其他节点结束的弧的数量是节点 x 的出度，并且对于每个节点 $y \in N$，$|\{x \mid (x, y) \in A\}|$，A 中在某个节点处开始并在 y 处结束的弧的数量是节点 y 的入度。节点 $n \in N$ 的度是 n 的入度和出度之和。

[0208] DG $G = (N, A)$ 中的源节点是其入度为零的节点，并且 DG $G = (N, A)$ 中的宿节点是其出度为零的节点。

[0209] DG $G = (N, A)$ 是控制流图(CFG)，如果且仅如果其具有特异源节点 $\mathbf{n_0} \in N$，从该特异源节点每个节点 $n \in N$ 是可达的。

[0210] 令 $G = (N, A)$ 是具有源节点 n_0 的 CFG。如果且仅如果每个从 n_0 开始并以 y 结束的路径包含 x，节点 $x \in N$ 支配节点 $y \in N$。(注意，按照该定义和上文的注释，每个节点支配其自己。)如果且仅如果以从开始节点开始并以 Y 的元素结束的每个路径包含 X 的元素，节点 X 的集合支配 CFG 中的节点 Y 的集合。

[0211] 在如上的$G = (N, A)$和s的情况下，如果且仅如果从n_0开始并在 Y 的元素结束的每个路径包含 X 的元素，非空节点集合$X \subseteq N$支配非空节点集合$Y \subseteq N$。(注意，单个节点支配另一单个节点的情况是该定义的特殊情况，其中 |X|=|Y|=1。)

2.2 代数结构。Z 表示所有整数的集合，并且 N 表示所有大于零的整数(自然数)的集合。$\mathbb{Z}/(m)$表示整数模 m 的环，对于某个整数，$m > 0$。只要 m 是素数，$\mathbb{Z}/(m) = \mathrm{GF}(m)$，整数模 m 的伽罗瓦域。B 表示比特的集合，其可以用环$\mathbb{Z}/(2) = \mathrm{GF}(2)$的两个元素来标识。

[0212] 2.2.1 恒等式。恒等式(即，等式)在混淆中扮演关键角色：如果对于两个表达式 X、Y，我们知道 X=Y，则我们可以用 Y 的值来替代 X 的值，并我们可以用 Y 的计算来替代 X 的计算，反之亦然。

[0213] 这样的基于代数恒等式的替代对于混淆是关键的容易通过以下事实看出：它们的使用被发现在$\{5, 7, 8, 10, 11, 12, 21, 22, 23, 24, 28, 29, 30\}$的每一个中改变外延。

[0214] 有时我们希望使布尔表达式恒等(相等)，这些布尔表达式可以自己包含等式。例如，在典型的计算机算法中，$x = 0 \text{ iff } (-(x \vee (-x)) - 1) < 0$(使用带符号的比较)。因此，“iff”等于条件，并且因此包含“iff”的表达式也是恒等式——具体地，条件恒等式或布尔恒等式。

[0215] 2.2.2 矩阵。我们通过以下来表示$r \times c$(r 行、c 列)矩阵 M：

$$M = \begin{bmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,c} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,c} \\ \vdots & \vdots & \ddots & \vdots \\ m_{r,1} & m_{r,2} & \cdots & m_{r,c} \end{bmatrix}$$

其中，其转置通过 M^T 来表示，其中

$$M^{\mathsf{T}} = \begin{bmatrix} m_{1,1} & m_{2,1} & \cdots & m_{r,1} \\ m_{1,2} & m_{2,2} & \cdots & m_{r,2} \\ \vdots & \vdots & \ddots & \vdots \\ m_{1,c} & m_{2,c} & \cdots & m_{r,c} \end{bmatrix}$$

使得，例如

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}^{\mathsf{T}} = \begin{bmatrix} a & c & e \\ b & d & f \end{bmatrix}。$$

[0216] 2.2.3 $\mathbb{Z}/(2^n)$与计算机算法的关系。在$\mathbb{B}^n$(所有长度为 n 的比特矢量的集合)上，像往常一样定义加法(+)和乘法$(\cdot)$定义以用于具有 2 的补码定点运算的计算机(参见 [25])。然后，$(\mathbb{B}^n, +, \cdot)$是阶 2^n 的有限 2 的补码环。模整数环$\mathbb{Z}/(2^n)$与$(\mathbb{B}^n, +, \cdot)$是同构的，其是在具有 n 比特字长的计算机上的典型计算机定点计算(加法、减法、乘法、除法和余数)

的基础。

[0217] (为了方便,我们可以通过 xy 来写$x \cdot y$(x 乘以 y);即,我们可以通过并列来表示乘法,这是代数中的公共约定。)

鉴于该同构,我们可互换地使用这两个环,尽管我们可以将$(\mathbb{B}^n, +, \cdot)$视为包含范围从-2^{n-1}到$2^{n-1}-1$(包含性的)中的带符号数。我们可以得以忽略$(\mathbb{B}^n, +, \cdot)$的元素是否占据带符号范围以上或量值从 0 到2^n-1(包含性的)的范围的问题的原因是算术运算"+"和"·"对 B^n 中的比特矢量上的影响是相同的,无论我们是将数解释为 2 的补码带符号数还是解释为二进制量值无符号数。

[0218] 我们是否将数解释为带符号的问题仅对于不等式运算$<, >, \leq, \geq$会出现,这意味着我们应当预先决定具体数要被如何处理:不一致的解释将产生异常结果,正如 C 或 C++ 编译器不正确的使用带符号和无符号比较指令将产生异常代码一样。

[0219] 2.2.4 按比特计算机指令和$(\mathbb{B}^n, \vee, \wedge, \neg)$。在$\mathbb{B}^n$(所有长度为 n 的比特矢量的集合)上,具有 n 比特字的计算机典型地提供按比特与($\wedge$)、含或($\vee$)以及非($\neg$)。于是$(\mathbb{B}^n, \vee, \wedge, \neg)$是布尔代数。在$(\mathbb{B}, \vee, \wedge, \neg)$中,其中矢量长度是一,0 为假且 1 为真。

[0220] 对于任何两个矢量$u, v \in \mathbb{B}^n$,我们通过$u \oplus v = (u \wedge (\neg v)) \vee ((\neg u) \wedge v)$定义 u 和 v 的按比特异或($\oplus$)。为了方便,我们典型地用$\bar{x}$表示$\neg x$。例如,我们还可以将该恒等式表示为$u \oplus v = (u \wedge \bar{v}) \vee (\bar{u} \wedge v)$。

[0221] 因为在布尔代数中的矢量乘法——按比特与($\wedge$)——是结合的,所以$(\mathbb{B}^n, \oplus, \wedge)$是环(称为布尔环)。

[0222] 2.2.5 T 函数和非 T 函数。在以下情况下从 w 比特字的 k 矢量映射到 w 比特字的 m 矢量的函数$f: (\mathbb{B}^w)^k \mapsto (\mathbb{B}^w)^m$是 T 函数:如果对于每对矢量$x \in (\mathbb{B}^w)^k, y \in (\mathbb{B}^w)^m$:-$y = f(x)$(其中$x' \neq x$且$y' = f(x')$,并且其中在 w 比特字中比特从 0 到 w-1 编号),则在其处 y 和 y' 不同的单元字中的最低编号的比特不比在其处 x 和 x' 不同的单元字中的最低编号的比特低。典型地,我们认为字内的这种编号为从低阶(2^0)到高阶(2^{w-1})比特,认为字表示二进制量值,所以我们可以将此重新声明为:输出比特可以仅取决于相同或更低阶的输入比特。

[0223] B^w 上计算的可从$\wedge, \vee, \oplus, \neg$连同在$\mathbb{Z}/(2^w)$上的$+, -, \times$组成的函数——从而在 w 比特字上的所有运算操作——是 T 函数。具有 T 函数属性的模糊构造易受比特切片攻击,因为我们可以通过将来自所有字的高阶比特掉入到输入和输出矢量中来从任何 T 函数获得另一合法的 T 函数。

[0224] T 函数属性不适于不是 2 的幂的右移位、按比特旋转、除法运算或基于除数 / 模数的余数 / 模运算,也不适于其中条件分支做决定的函数,其中较高阶条件比特影响较低阶

输出比特的值。

[0225] 对于条件分支和基于比较的条件执行，注意：基于使用六种标准比较 $=, \neq, <, >, \leq, \geq$ 形成的条件的条件执行均可以容易地违反T函数条件，并且实际上，在使用基于比较的分支逻辑的正常代码中，相对于服从T函数，违反T函数更加容易。

[0226] 2.2.6 多项式。多项式是形式为 $f(x) = \sum_{i=0}^{d} a_i x^i = a_d^d + \cdots + a_2 x^2 + a_1 x + a_0$（其中对于任何x，$x^0 = 1$）。如果 $a_d \neq 0$，则d是多项式的次数。多项式可以相加、相减、相乘和相除，并且这样的运算的结果本身是多项式。如果d=0，则多项式是常数；即，其简单地由标量常数 a_0 构成。如果d>0，则多项式是非常数。我们可以具有在有限和无限环和域上的多项式。

[0227] 如果非常数多项式不能写为两个或更多非常数多项式的乘积，则该非常数多项式是不可约的。不可约的多项式对于多项式所扮演的角色类似于素数对于整数所扮演的角色。

[0228] 变量x没有特殊意义：关于特定多项式，其仅仅是占位符。当然，我们可以将x替代为一值以评估多项式——即，变量x仅在我们将其替代为某内容时才是有意义的。

[0229] 我们可以用其系数 $(d+1)$ 矢量 $(a_d, \ldots, a_1, a_0)$ 来标识多项式。

[0230] 在 $GF(2) = \mathbb{Z}/(2)$ 上的多项式在密码学中具有特殊意义，因为系数的 $(d+1)$ 矢量简单地是比特串并且可以在计算机上高效地被表示（例如，高达7次的多项式可以表示为8比特字节）；加法和减法是相同的；以及比特串表示的两个这样的多项式之和使用按比特 $\oplus$（异或）来计算。

[0231] 2.3 编码。我们在此正式地引入编码。

[0232] 令 $F :: D \mapsto R$ 是总计。选择双射 $d :: D \mapsto D'$ 和双射 $r :: R \mapsto R'$。我们称 $F' = r \circ F \circ d^{-1}$ 为F的编码版本。d是输入编码或定义域编码并且r是输出编码或值域编码。诸如d或r的双射简称编码。在特定情况中，F是函数。

[0233] 图1中示出的图然后交换，并且用F'的计算简单地是用加密的函数的计算[28,29]。如图1中所示，仅D'、F'（加密的函数）和R'对于攻击者是可见的。没有原始信息对于攻击者是可见的（D、F、R），也没有原始信息是用于执行编码的信息。

[0234] 令 $B_i :: S_i^{m_i} \mapsto S_i^{n_i}$，其中对于 $i = 1, 2, \ldots, k$，$m_i, n_i \in \mathbf{N}$。然后，关系级联 $B_1 \| B_2 \| \ldots \| B_k$ 是关系 $B :- \forall v_i \in S_i^{m_i}, v_i' \in S_i^{n_i}$，i范围在 $\{1, 2, \ldots, k\}$ 上，$(v_i, v_i') \in B_i,\ i = 1, 2, \ldots, k$ **iff** $(v_1 \| \ldots \| v_k,\ v_1' \| \ldots \| v_k') \in B$。明显的，$B^{-1} = B_1^{-1} \| B_2^{-1} \| \ldots \| B_k^{-1}$。如果 $B_1, \ldots, B_n$ 是双射并因此是编码，则B也是双射和编码。B于是称为级联编码，并且 B_i 是B的第i个分量。

[0235] （我们可以将以下视为以上的特殊情况，其中 m_i 和 n_i 都具有值1。）对于 $i = 1, 2, \ldots, k$，令 $B_i :: S_i \mapsto S_i'$。然后，关系聚合 $[B_1, B_2, \ldots, B_k]$ 是关系

$B :- \forall x_i \in S_i, x'_i \in S'_i$，其中 i 范围在$\{1,2,\ldots,k\}$上，

$$(x_i, x'_i) \in B_i,\ i = 1,2,\ldots,k \quad \text{iff} \quad ((x_1,\ldots,x_k),\ \langle x'_1,\ldots,x'_k\rangle) \in B.^1$$

。明显的，$B^{-1} = \{B_1^{-1},\ldots,B_k^{-1}\}$。如果$B_1,\ldots,B_n$是双射并因此是编码，则 B 也是双射和编码。B 于是称为聚合编码，并且 B_i 是 B 的第 i 个分量。

[0236] 对于$i = 1,2,\ldots,k$，令$B_i :: S \rightarrowtail S'_i$。则关系凝聚$\{B_1; B_2; \ldots; B_k\}$是关系$B :- \forall x'_i \in S'_i, \forall x \in S, (x,(x'_1,\ldots,x'_k)) \in B$ iff $((x,\ldots,x),(x'_1,\ldots,x'_k)) \in (B_1,\ldots,B_k)$。

[0237] 2.3.1 网络编码计算。一般地，变换的输出将成为另一后续变换的输入，这意味着第一个的输出编码必须匹配如下那样匹配第二个的输入编码。

[0238] 用于计算$Y \circ X$的联网编码(即，变换 X 被变换 Y 跟随)是形式$Y' \circ X' = (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) = H \circ (Y \circ X) \circ F^{-1}$的编码。

[0239] 在一般情况中，我们具有这样的编码网络，其是其中节点函数是经编码的函数的数据流网络。

[0240] 编码可以从代数结构得到(见§2.2)。例如，有限环编码(FR)基于这样的事实：$\mathbb{Z}/(2^w)$上的仿射函数$x' = e_x(x) = sx + b$只要 s 是奇数就是无损的，其中 w 是字宽，其可以通过忽略溢流使得模数是自然机器整数模数来实现。

[0241] 我们从图 1 注意到：对于经编码的计算的关键是输入、输出以及计算都被编码。例如，考虑$\mathbb{Z}/(2^w)$，整数的环模2^w，其中 w 是用于某些计算机的优选字宽(典型地为 8、16、32 或 64，随着时间过去趋势朝向更高的宽度)。$\mathbb{Z}/(2^w)$的单位(即，具有乘法逆元素的那些)是奇元素$1,3,5,\ldots,2^w-1$。

[0242] 假设我们想要在字宽为 w 的二进制计算机上对加法、减法和乘法进行编码，使得未编码的计算在$\mathbb{Z}/(2^w)$上执行。我们可以使用在$\mathbb{Z}/(2^w)$上的仿射编码。对于未编码变量x,y,z和对应的经编码变量x',y',z'，其中$x' = e_x(x) = s_x x + b_x \quad y' = e_y(y) = s_y y + b_y \quad z' = e_z(z) = s_z z + b_z$，我们想要确定如何计算

---

[1] $[f_1,\ldots,f_k]$记号由 John Backus 在他的 ACM 图灵奖演讲中针对函数聚合引入。我已经将其应用于一般性的二元关系。

[0243] $z' = x' +' y' \quad z' = x' -' y' \quad z' = x' \times' y'$；即，我们需要对$+', -', \times'$的表示。(在这样的运算的网络上，我们将具有许多不同的编码$+', -', \times'$，其中要求运算的结果采用与消耗运算的对应输入编码相同的编码。)

在并列在$\mathbb{Z}/(2^w)$上表示$\times$，2 的补码中的 -x 是$\mathbb{Z}/(2^w)$中的 -x，并且 2 的补码中的 xy 是$\mathbb{Z}/(2^w)$中的 xy。因此，如果e_v是 v 的编码并且e_v^{-1}为其逆，则$v' = e_v(v) = s_v v + b_v$并且$e_v^{-1}(v') = (v' - b_v)s_v^{-1} = s_v^{-1}v' + (-b_v s_v^{-1})$($\mathbb{Z}/(2^w)$上的另一仿射)。于是

$$\begin{aligned} z' &= x' +' y' \\ &= e_z(e_x^{-1}(x') + e_y^{-1}(y')) \\ &= (s_x^{-1}s_z)x' + (s_y^{-1}s_z)y' + (b_z - s_x^{-1}s_z b_x - s_y^{-1}s_z b_y) \end{aligned}$$

其具有一般形式$z' = c_1x' + c_2y' + c_3$，具有常数c_1, c_2, c_3；原始数据和编码系数已经消失。如果 y 是正或负常数 k，则我们可以选择$e_y = \mathbf{id}$(即，$s_y = 1$且$b_y = 0$)，其上式约减为

$$\begin{aligned} z' &= x' +' k \\ &= e_z(e_x^{-1}(x') + k) \\ &= (s_x^{-1}s_z)x' + (b_z - s_x^{-1}s_z b_x + s_z k) \end{aligned}$$

其对于常数c_1, c_2具有一般形式$z' = c_1x' + c_2$。可替代地，我们可以计算$z' = x' +' k$为$z' = x'$，其中我们定义$s_z = s_x$并且$b_z = b_x - s_x k$，使得我们可以在完全没有计算的情况下计算$z' = x' +' k$。为了在没有计算的情况下使$z' = -'x'$，我们简单地定义$s_z = -s_x$和$b_z = -b_x$以及设置$z' = x'$。类似地，对于减法：

$$\begin{aligned} z' &= x' -' y' \\ &= e_z(e_x^{-1}(x') - e_y^{-1}(y')) \\ &= (s_x^{-1}s_z)x' + (-s_y^{-1}s_z)y' + (b_z - s_x^{-1}s_z b_x + s_y^{-1}s_z b_y) \end{aligned}$$

其再次具有一般形式$z' = c_1x' + c_2y' + c_3$，具有常数c_1, c_2, c_3；原始数据和编码系数已经消失。如果y'是常数 c，则我们如上针对加法继续，设置$k = -c$。为了相减$z' = k -' x'$，我们可以通过在没有计算的情况下忽略x'而在没有计算的情况下计算它，并且然后如上所描述的那样加 k。对于乘法：

$$\begin{aligned} z' &= x' \times' y' \\ &= e_z(e_x^{-1}(x') \times e_y^{-1}(y')) \\ &= (s_x^{-1}s_y^{-1}s_z)x'y' + (-s_x^{-1}s_y^{-1}s_z b_y)x' \\ &\qquad + (-s_x^{-1}s_y^{-1}s_z b_x)y' + (b_z + s_x^{-1}s_y^{-1}s_z b_x b_y) \end{aligned}$$

其对于常数c_1, c_2, c_3, c_4具有一般形式$z' = c_1x'y' + c_2x' + c_3y' + c_4$。最后，如果 x 是常数 k，则我们可以选择$e_x = \mathbf{id}$(即，我们可以选择$s_x = 1$和$b_x = 0$)，在该情况中，以上乘法方程约减为：

$$\begin{aligned} z' &= k \times' y' \\ &= e_z(k \times e_y^{-1}(y')) \\ &= (s_y^{-1}s_z k)y' + (b_z - s_y^{-1}s_z k b_y) \end{aligned}$$

其对于常数c_1, c_2具有一般形式$z' = c_1y' + c_2$。可替代地，如果 k 在$\mathbf{Z}/(2^w)$中是可逆的，则我们可以通过定义$s_z = k^{-1}s_y$和$b_z = b_y$将$z' = k \times' y'$计算为$z' = y'$，其具有用于 FR 编

码的标准仿射形式，并且允许我们取y'为z'，但是是利用编码e_z而非其自己的编码e_y，从而我们可以在完全没有计算的情况下计算$z' = k \times' y'$。

[0244] 也可以使用较高阶的多项式：一般而言[27]，对于$1 < w \in \mathbb{N}$，在$\mathbb{Z}/(2^w)$上，$P(x) = a_0 \diamond_1 a_1 x \diamond_2 a_2 x^2 \diamond_3 \ldots \diamond_d a_d x^d$是置换多项式(即，双射或无损多项式)iff

(1)对于$i = 1, \ldots, d$，$\diamond_i$是$+$(模2^w)，

(2)a_1是奇数，

(3)$a_2 + a_4 + \ldots$是偶数，以及

(4)$a_3 + a_5 + a_7 + \ldots$是偶数；

特性归因于Rivest。(仅$\mathbb{Z}[0, 2^w - 1]$上的双射的集合可以写为在$\mathbb{Z}/(2^w)$上的这样的置换多项式。)次数越高，在多项式的选择中包含更多熵，但是时间和空间复杂度相应增加。

[0245] 置换多项式在许多环和域上存在。Klimov[17]将该特性外延到他称为一般化置换多项式的内容，其类似于上文描述的那些，以下除外：任何给出的$\diamond_i$可以是$\mathbb{B}^w$上的+或-(模2^w)或按比特异或$(\oplus)$，并且$\diamond$运算可以以任何固定顺序来应用。

[0246] 尽管我们可以写任意次数的多项式，但是在$\mathbb{Z}/(2^w)$上的每个多项式等同于非常受限次数的多项式。事实上，已知的是，对于$w \in \mathbb{N}$，在$\mathbb{Z}/(2^w)$上的每个置换多项式P具有次数$\leq w + 1$的在$\mathbb{Z}/(2^w)$上的等同置换多项式Q。

[0247] 关于置换多项式的难度(无论一般化与否)是它们仅在它们的逆已知并且方便计算时才变得真正有用。已知大多数置换多项式具有高次数的逆(对于在$\mathbb{Z}/(2^w)$上的置换多项式接近2^w)。然而，使用Rivest的(非一般化的)以上特性，如果对于$i = 2, \ldots, d, a_i^2 = 0$，则逆的次数与要求逆的多项式的次数相同。针对置换多项式的逆的方程如下(对于更严格的处理参见第C节)：

2.3.2 二次多项式和逆。如果$P(x) = ax^2 + bx + c$，其中$a^2 = 0$并且b是奇数，则P是可逆的，并且$P^{-1}(x) = dx^2 + ex + f$，其中常数系数通过以下来定义：

$$d = -\frac{a}{b^3},$$

$$e = 2\frac{ac}{b^3} + \frac{1}{b},$$ 以及

$$f = -\frac{c}{b} - \frac{ac^2}{b^3}。$$

[0248] 2.3.3 三次多项式和逆。如果$P(x) = ax^3 + bx^2 + cx + d$，其中$a^2 = b^2 = 0$且c是奇数，则P是可逆的，并且$P^{-1}(x) = ex^3 + fx^2 + gx + h$，其中常数系数通过以下来定

义：

$$c = -\frac{a}{c^4},$$

$$f = 3\frac{ad}{c^4} - \frac{b}{c^3},$$

以及

$$g = \frac{1}{c} - 6\frac{ad^2}{c^4} + 3\frac{ad^2}{c^4} + 2\frac{bd}{c^3},$$

$$h = -ed^3 - \left(3\frac{ad}{c^4} - \frac{b}{c^3}\right)d^2 - \left(\frac{1}{c} - 6\frac{ad^2}{c^4} - 3d^2e + 2\frac{bd}{c^3}\right)d。$$

[0249] 2.3.4 四次多项式和逆。如果 $P(x) = ax^4 + bx^3 + cx^2 + dx + e$，其中 $a^2 = b^2 = c^2 = 0$ 并且d是奇数，则P是可逆的并且 $P^{-1}(x) = fx^4 + gx^3 + hx^2 + ix + j$，其中常数系数通过以下来定义：

$$f = -\frac{a}{d^5},$$

$$g = \frac{4ae}{d^5} - \frac{b}{d^4},$$

以及

$$h = -6\frac{ae^2}{d^5} + 3\frac{be}{d^4} - \frac{c}{d^3},$$

$$i = \frac{4ae^3}{d^5} - 3\frac{be^2}{d^4} + 2\frac{ec}{d^3} + \frac{1}{d},$$

$$j = -\frac{ae^4}{d^5} + \frac{be^3}{d^4} - \frac{ce^2}{d^3} - \frac{e}{d}。$$

[0250] 2.3.5 关于在 $\mathbb{Z}/(p^w)$ 上的置换多项式的注意事项。令p是素数并且 $w \in \mathbb{N}$。在 $\mathbb{Z}/(p^w)$ 上的属性置换多项式在Mullen和Stevens的1984年的论文中被考察[19]，其向我们教导如下。

[0251] （1）在 $\tau(m)$ 是 $\mathbb{Z}/(m)$ 上的PPs的数量的情况下，对于任意 $m \in \mathbb{N}$，m>0，其中 $m = \prod_{i=1}^{k} p_i^{n_i}$，$p_1, \ldots, p_k$ 是不同素数且 $n_1, \ldots, n_k \in \mathbb{N}$，我们有 $\tau(m) = \prod_{i=1}^{k} \tau(p_i^{n_i})$。

[0252] （2）在 $\mathbb{Z}/(p^n)$ 上的功能上不同的PPs的数量为 $\tau(p^n) = p^{n(\mathcal{N}_p(n)+1)-\mathcal{S}_p(n)}$，其中 $\mathcal{S}_p(n) = \sum_{i=0}^{\mathcal{N}_p(n)} \nu_p(i)$，$\mathcal{N}_p(n)$ 是最大整数 $\rho :- \nu_p(\rho) \leq n$，并且 $\nu_p(s) = \sum_{i=1}^{\infty} \left\lfloor \frac{s}{p^i} \right\rfloor$。注意，$\mathcal{N}_p(n)$ 是整数 $\kappa :- p^n \mid (\kappa+1)!$ 但是 $p^n \nmid \kappa!$，并且我

们有 $\mathcal{N}_p(n) \equiv -1$(模 p)。

[0253] (3)每个多项式函数 $f(x) = \sum_{i=1}^{n} a_i x^i$ 可以以递降阶乘的形式来表达

$$f(x) = \sum_{i=0}^{\mathcal{N}(n)} c_i x^{(i)}$$

其中 $x^{(i)} = \prod_{i=0}^{i-1}(x-i)$，$x^{(0)} = 1$。$x^{(i)}$ 是递降阶乘。

[0254] 2.3.6 关于在 $\mathbb{Z}/(2^w)$ 上的置换多项式的注意事项。对于具有 w 比特字的计算机，在 $\mathbb{Z}/(2^w)$ 上的 PPs 是特别方便的，因为加法、乘法以及减法模 2^w 可以通过以下方式来执行：简单地忽略溢流、下溢以及在量值和 2 的补码计算之间的不同，并针对这样的机器取普通未校正的硬件结果。此外，这也是在用 C、C++ 或 Java™ 编写的程序中对整型、带符号整型或无符号整型的操作数的计算的默认行为。

[0255] 调整来自以上的结果[19]，在 $\mathbb{Z}/(2^w)$ 上功能上不同的 PPs 的数量为：$\tau(2^w) = 2^{w(\mathcal{N}_2(w)+1)-\mathcal{S}_2(w)}$，其中 $\mathcal{S}_2(w) = \sum_{i=0}^{\mathcal{N}_2(w)} \nu_2(i)$，$\mathcal{N}_2(w)$ 是最大整数 $\rho :- \nu_2(\rho) < w$，并且 $\nu_2(s) = \sum_{i=1}^{\infty} \left\lfloor \frac{s}{2^i} \right\rfloor$。注意，$\mathcal{N}_2(n)$ 是整数 $\kappa :- 2^n \mid (\kappa+1)!$ 但是 $2^n \nmid \kappa!$，并且我们有 $\mathcal{N}_2(n) \equiv -1 \equiv 1$ (模 2)。

[0256] 2.3.7 关于编码的一般注意事项。P' 表示从函数 P 得到的经编码的实现。为了强调 P 将 m 矢量映射到 n 矢量，我们写 ${}^{n}_{m}P$。P 于是称为 $n \times m$ 函数或 $n \times m$ 变换。对于矩阵 M，${}^{n}_{m}M$ 指示 M 具有 m 列和 n 行。(这些记号自然地对应，将 M 到示例的应用取为函数应用。)。

[0257] ${}^{n}_{m}\mathbf{E}$ (助记符：熵传递函数)是从在 B 上的 m 矢量到在 B 上的 n 矢量的任意函数，对于 $m \le n$，其不丢失信息的比特，并且对于 $m > n$，其最多丢失 m-n 比特。函数 ${}^{n}_{n}f$ 是有损的，其不是 ${}^{n}\mathbf{E}$ 的实例。在给定方程或等式中的 ${}^{n}_{m}\mathbf{E}$ 的多次出现表示相同矢量。

[0258] ${}_{n}\mathbf{C}$ (助记符：熵矢量)是从 $\mathbb{B}^n$ 选择的任意矢量。${}_{n}\mathbf{C}$ 在给定方程或等式中的多次出现表示相同矢量。

[0259] 仿射函数或仿射变换(AT)是针对所有矢量 ${}_{m}v \in S^m$ 针对所有集合 S 通过 ${}^{n}_{m}V({}_{m}v) = {}^{n}_{m}M_m v + {}_{n}d$ 定义的矢量到矢量函数 V(简明地：$V(x) = Mx + d$)，其中 M 是常数矩阵，并且 d 是常数位移矢量。如果 A 和 B 是 AT，则其中定义的 $A \vec{\|} B$，$[A, B]$ 和 $A \circ B$ 也是。AT $V(x) = Mx + d$ 是线性函数或线性变换(LT) iff $d = (0)^n$。

[0260] 对于一些素数幂 ξ 的在 $\{F, +, \cdot\} \in \mathrm{GF}(\xi)$ 上的从 k 矢量到 m 矢量的函数 $f: F^k \mapsto F^m$ 是深度非线性的，iff 对 线性函数 $g: F^k \mapsto F^m$ 和编码

$d_1,\ldots,d_k,r_1,\ldots,r_m:F\mapsto F:-f=[r_1,\ldots,r_m]\circ g\circ[d_1^{-1},\ldots,d_k^{-1}]$。

[0261] （注意，如果$\exists g':-f=[r'_1,\ldots,r'_m]\circ g'\circ[d_1^{-1},\ldots,d_k^{-1}]$，其中$g'$是仿射，则无疑$\exists$线性$g,r_1,\ldots,r_m:-f=[r_1,\ldots,r_m]\circ g\circ[d_1^{-1},\ldots,d_k^{-1}]$，因为我们可以选择$r_1,\ldots,r_m$来执行$g'$的矢量位移的按元素的相加。）

如果对于素数幂$|A|>1$，$g:A^k\mapsto A^m$不是深度非线性的，我们说g直到I/O编码是线性的。

[0262] 我们已经证明了关于直到I/O编码的线性度和恒等式的以下内容。

[0263] （1）如果函数直到I/O编码是线性的，则所有其射影也是。

[0264] （2）两个矩阵直到I/O编码是相同的，iff一个可以通过一系列的行或列与非零标量的乘法而转换成另一个。

[0265] （3）如果两个函数直到I/O编码是线性的并且直到I/O编码是相同的，则它们是矩阵的I/O编码，所述矩阵直到I/O编码也是相同的。

[0266] （4）如果两个矩阵直到I/O编码是相同的，则它们的相应子矩阵也是。

[0267] （5）如果M在GF(n)上是非零矩阵，则在GF(n)上存在矩阵M’使得M、M’直到I/O编码是相同的，其中M’（M的I/O规范形式）具有仅包含0和1的前导行和列。

[0268] 2.3.8 *分式和分式函数。如§2.3.1中所提到的，一般而言，当产生经编码的输出时，其与假设的精确相同的编码一起消耗，使得经编码的运算$z=f(x,y)$变成$z'=f'(x',y')$，其中$(x',y',z')=(e_x(x),e_y(y),e_z(z))$，$e_x,e_y,e_z$是编码且$f'=e_z\circ f\circ[e_x^{-1},e_y^{-1}]$。

[0269] 有时有利的是输出具有一个编码的值且随后输入假设一些其他编码。如果我们将x输出为$e_1(x)$并且之后消耗它假设编码e_2，则实际上，我们已将$e_2^{-1}\circ e_1$应用于未编码的值。我们称在其中产生值的编码与当其被消耗时假设的编码之间的这样的有意失配称为分式。如果编码是线性的，则分式函数$e_2^{-1}\circ e_1$也是，并且如果它们是置换多项式，则分式函数$e_2^{-1}\circ e_1$也是。

[0270] 分式在混淆中是潜在有用的，因为它们有效执行的计算在代码中不出现——执行正常联网编码的代码和借助于分式增加运算的代码的量和形式——是相同的，并且不存在明显的方式来消除这些情况的歧义，因为对它们自身的编码趋向于是稍微歧义的。

[0271] 注意，定义分式的属性是所说的分式函数$v^{-1}\circ u$。一般而言，存在对消耗编码v和产生编码u的许多不同选择，这精确地产生相同的分式函数：很可能例如具有$u_1,\ldots,u_k,v_1,\ldots,v_k$使得$v_i^{-1}\circ u_i$对于$i=1,\ldots,k$是相同的分式函数。因此，指定分式函数未明确隐含其的产生和消耗编码。

[0272] 2.4 部分评估(PE)。MF的部分评估(PE)是通过冻结一些其他MF（或这样生成的MF)的一些输入的MF的生成。更正式地，令$f:X\times Y\mapsto Z$是MF。针对常数$c\in Y$的对f的部

分评估(PE)是该MF的导数$g: X \rightarrow Z$，使得对于任何$x \in X$和$z \in Z$，$g(x) \rightarrow z$ iff $f(x,c) \rightarrow z$。为了指示该PE关系，我们还可以写$g(\cdot) = f(\cdot, c)$。我们还可以将MF对f的PE求导的g称为f的部分评估(PE)。即，术语评估可以用于指代求导过程或其结果。

[0273] 为了提供特定示例，我们考虑编译的情况。

[0274] 在没有PE的情况下，对于计算机程序p，我们可以具有$p: S \rightarrow E$，其中S是所有源代码文件的集合，并且E是目标代码文件的集合。于是，$e = p(s)$将表示计算机程序p到源代码文件s的应用，产生目标代码文件e。(我们取p为函数，并且不仅仅是多函数，因为我们通常希望编译器是确定性的。)

现在假设我们具有非常一般的编译器q，其输入源程序s以及一对语义描述：在期望目标平台t上的可执行代码的语义的描述和源语言语义描述d。其根据源语言语义描述将源程序编译成用于期望目标平台的可执行代码。于是我们有$q: S \times (D \times T) \rightarrow E$，其中S是源代码文件的集合，D是源语义描述的集合，T是平台可执行代码语义描述的集合，以及E是用于任何平台的目标代码文件的结合。于是，对于常数元组$(d,t) \in D \times T$(即，由特定源语言语义描述和特定目标平台语义描述构成的对)，特定编译器是q的PE p，即对于一些特定常数$(d,t) \in D \times T$，$p(s) = q(s,(d,t))$。在该情况下，X(PE保留的输入集合)是S(源代码文件的集合)，Y(PE通过选择其特定成员而移除的输入集合)是$D \times T$(源语义描述的集合D和目标平台语义描述的集合T的笛卡尔积)，以及Z(输出集合)是E(目标代码文件的集合)。

[0275] PE在[10, 11]中使用：AES-128密码[16]和DES密码[18]是针对密钥的部分评估以便隐藏密钥免受攻击者影响。对基本方法和系统的更详细的描述在[21, 22]中给出。

[0276] 当优化编译器通过确定操作数在运行时将在何处是常数以及然后用更特定的运算代替它们与常数的运算来用更特定的计算替代一般计算时，优化编译器执行PE，所述更特定的运算不再需要输入(有效地恒定的)操作数。

[0277] 2.5 输出外延(OE)。假设我们具有函数$f: U \rightarrow V$。函数$g: U \rightarrow V \times W$是$f$的输出外延(OE)，iff对于每个$u \in U$，我们对于一些$w \in W$有$g(u) = (f(u), w)$。即，$g$向我们给出$f$给出的任何内容，并且还给出输出信息以外的附加过程。

[0278] 我们还使用术语输出外延(OE)来指代给定这样的函数f找到这样的函数g的过程。

[0279] 在函数f实现为例程或其他程序段的情况下，一般直接确定实现作为函数f的OE的函数g的例程或程序段，因为找到这样的函数g的问题受非常松的约束。

[0280] 2.6 逆向部分评估(RPE)。为了创建一般的、低开销的、有效的互锁以将保护绑定到SBE，我们将采用新颖的基于逆向部分评估(RPE)的方法。

[0281] 明显地，对于几乎任何MF或程序$g: X \rightarrow Z$，存在极大的程序或MF f的集合(集合Y)以及常数$c \in Y$，对于所述集合，针对任何任意的$x \in X$，我们始终具有$g(x) = f(x,c)$。

[0282] 我们称找到这样的元组(f,c,Y)的过程(或我们通过该过程找到的元组)为g的逆

向部分评估(RPE)。

[0283] 注意,PE 趋向于是特定的且确定性的,而 RPE 提供无定数的大量替代:对于给定g,可以存在任何数量的不同元组$\{f, c, \ldots\}$,这些元组中的每一个都有资格作为g的 RPE。

[0284] 找到作为更一般的程序的PE的有效程序可能是非常难的——即,该程序受非常紧的约束。找到给定特定程序的有效 RPE 正常情况下是很简单的,因为我们有如此多的合法选择——即,该问题受非常松的约束。

[0285] 2.7 代码编译中的控制流图(CFG)。在编译器中,我们典型地通过控制流图(CFG:参见§2.1.3中的定义)来表示通过程序的可能控制流,其中可执行代码的基本块(BB)(具有单个起始点、单个结束点并且从其起始点到结束点依次执行的“直线”代码序列)由图节点表示,并且弧将对应于BB U的节点连接到对应于BB V的节点,如果在包含程序的执行期间控制将时钟或可以可能地从 BB U 的结束流向 BB V 的起始的话。这可以以多种方式发生:

(1)控制流可以自然地从 BB U 导向 BB V。

[0286] 例如,在以下 C 代码段中,控制流自然地从 U 导向 V:

```
switch(radix) {
    case HEX:
            U
    case OCT:
            V
    ...
}
```

(2)控制流可以通过过程内控制构造从 U 引导到 V,所述过程内控制构造诸如 while 循环、if 语句或 goto 语句。

[0287] 例如,在以下 C 代码段中,控制通过 break 语句从 A 引导到 Z:

```
switch(radix) {
    case HEX:
            A
            break;
    case OCT:
            B
    ...
}
Z
```

(3)控制流可以通过调用或返回从 U 引导到 V。

[0288] 例如,在以下 C 代码段中,控制通过在 g() 体中调用到 f() 而从 B 引导到 A,以及通过从至 f() 的调用返回从 A 引导到 C:

```
void f(void) {
    A
    return;
}
int g(int a, float x) {
    B
    f();
    C
}
```

(4)控制流可以通过异常控制流事件从U引导到V。

[0289] 例如,在以下C++代码段中,控制通过指向所说的对类A中的对象的引用的引用的dynamic_cast的故障而可能从U引导到V:

```
#include<typeinfo>
...
int g(int a, float x) {
    ...
    try {
        ...
        U
        A& x = dynamic_cast<A&>(y);
        ...
    catch(bad_cast c) {
        V
    }
    ...
}
```

对于CFG $C=(N,T)$ 中的每个节点 $n \in N$ ——针对控制的C、针对传递的T——取节点n来表示特定BB,并且该BB计算通过节点确定的MF,所述BB n包含:某个函数 $f: X \to Y$,其中X表示由n的代码读取和使用的所有可能值的集合(并因此至函数 f 的输入),并且Y表示由n的代码写出的所有可能值的集合(并因此来自函数 f 的输出)。典型地,f 是函数,但是如果 f 利用非确定性的输入,诸如高分辨率硬件时钟的当前读数,则 f 是MF而非函数。此外,一些计算机硬件包括可以产生非确定性结果的指令,这再次可以导致 f 是MF而非函数。

[0290] 对于具有CFG $C=(N,T)$ 和起始节点 n_0 的整个程序,我们用程序的BB的集合来标识N,我们用出现在程序的起始点处的BB(典型地,对于C或C++程序的例程main()的起始BB)来标识 n_0,并且我们用控制从程序的一个BB到另一个的每个可行传递来标识T。

[0291] 有时,取代用于整个程序的CFG的是,我们可以具有用于单个例程的CFG。在该情况中,我们用例程的BB来标识N,我们用出现在例程的起始处的BB来标识 n_0,并且我们用控制从例程的一个BB到另一个的每个可行传递来标识T。

[0292] 2.8 通过对交互进行的*置换。这里我们考虑如何仅使用随机的 2×2 切换元素来产生n个元素的置换,其计算 $y_1,y_2 \leftarrow x_1,x_2$ (无交换)或 $y_2,y_1 \leftarrow x_1,x_2$ (交换),分别具有

$\frac{1}{2}$概率。

[0293] 2.8.1 通过盲对交换进行的＊置换。排序网络可以如图 7 中示出的那样通过一系列平行线 72 来表示，其中，在某些点处，在与这些线的直角处，一个线通过交叉连接 74（表示对作为两个连接的线携带的元素的数据的比较 - 并且如果更大 - 则交换操作）连接到另一个线。如果不管输入为何，输出以排序后的顺序出现，则产生的网络是排序网络。在排序网络中的比较是数据无关的：正确的排序结果在线的右端，而不管线的左端处引入的数据。比较 - 并且如果更大 - 则交换操作可以被重排序，只要共享端点的比较的相对顺序被保持即可。

[0294] 构造这样的针对 n 个节点的排序网络的有效方式由 Batcher 的奇偶合并排序 [2] 给出，其是数据无关的排序：确切的说，相同的比较 - 并且如果更大 - 则交换操作被执行，而不管要排序的数据。在对 n 个元素的集合进行排序时，该算法执行$O(n(\log n)^2)$比较。算法的细节可以在这些 URL[3] 处找到。

[0295] 如果这样的网络将要把任意的数据排序成有序的，则其遵循：如果比较 - 并且如果更大 - 则交换操作被具有$\frac{1}{2}$概率的交换操作替代，则相同的网络配置将把 n 个独特元素的序列置换成随机顺序，可能是偏置的顺序。这是我们将用来使用伪随机真 / 假变量来实现置换的机制的基础，所述伪随机真 / 假变量是使用计算来创建的。

[0296] 注意，n 个元素的置换的数量是 n ！，而使用位置2^i中的一个比特来指示是进行还是不进行第 i 交换的交换配置的数量是$2^{b(n)}$，其中$b(n)$是在对 n 个元素排序的 Batcher 网络中的阶段的数量。

[0297] 例如，对于 n=3，n ！ =6，b(n)=3，以及$2^{b(n)} = 8$，所以必然存在比一种方式更多的可以选择的置换，但是它们中的一些不能。类似地，对于$n = 5,\ n! = 120,\ b(n) = 9$以及$2^{b(n)} = 512$，所以必然存在比一种方式更多的可以选择的置换，但是选择给定置换的交换配置的数量不能始终相同，因为$120 \nmid 512$。

[0298] 减小偏置要求我们确保达到任何置换的方式的数量对于每个置换大致相同。因为$2^{b(n)}$对于任何 n 都是二的幂，所以这不能简单地通过添加额外阶段来完成。在一些情况中，有必要使用用于减小偏置的其他方法。

[0299] 2.8.2 通过受控对交换进行的＊置换。在§2.8.1中描述的方法遭受显著的偏置（该偏置可以容易地超过二对一）。问题是：随机交换 / 不交换决定的数量始终是 2 的幂，而置换的数量始终是阶乘，并且对于 2 以上的元素计数，置换的数量决不会除尽交换 / 不交换队列的数量，该交换 / 不交换队列的数量形成介于 0 与$2^k - 1$之间的数（包含性的），其可以视为 k 比特的串：每交换 / 不交换决定一个比特。

[0300] 存在两种不同的机制我们可以部署来从相同种类的决定元素获得结果（比较两个伪随机数）。我们从直接选择问题开始：为了生成置换，我们针对给定位置选择 n 个元素之一，并且然后针对另一个位置选择 n-1 个元素之一，如此类推，直到我们被迫针对剩余的位置选择剩余的元素。

[0301] 移除偏置的第一种方法可以称为衰减。例如，假设我们需要选择 12 个元素之一。

我们可以创建具有 16 个叶子节点的二进制决策树，并且将这些叶子节点映射到 12 个元素上，其中 8 个经由一个叶子可达且 4 个通过两个叶子节点分别可达。（我们简单地将 16 个叶子节点卷绕 12 个选择，直到所有叶子节点已被使用；即，我们通过重复从 1 到 12 的元素号直到我们具有针对 16 个叶子节点中的每一个的元素号为止，来创建 16 个元素的序列：$(1,2,3,4,5,6,7,8,9,10,11,12,1,2,3,4)$。）在该点处，在我们的选择中我们具有最大 2:1 的偏置。如果我们使用具有 32 个叶子节点的树，则我们将具有 4 个从两个叶子节点可达的元素和 8 个从 3 个叶子节点可达的元素，并且我们的最大偏置减小到 3:2。如果我们使用具有 64 个叶子节点的树，则我们将具有 8 个从 5 个叶子节点可达的元素和 4 个从 6 个叶子节点可达的元素，并且我们的最大偏置减小到 5:4。衰减产生快速但庞大的代码。

[0302] 移除偏置的第二种方法可以称为重选。如上面那样，假设我们需要选择 12 个元素之一。我们可以创建具有 16 个叶子节点的二进制决策树，并且将 12 个叶子节点映射到 12 个选择。其他四个选择映射以向后循环并再次执行整个选择。以$\frac{3}{4}$概率，我们在第一次尝试时成功。以$\frac{15}{16}$的概率，我们在前两次尝试中成功。以$\frac{63}{64}$概率，我们在前三次尝试中成功。重选具有以下优点：其可以几乎完全消除偏置。其具有以下优点：在空间上紧致的同时，其涉及重做一些步骤。此外，因为我们将希望限制迭代的数量，所以其涉及对重复计数并默认成在其中计数被超过的这些稀少情况上的(稍)偏置的选择。重选是更紧致的、较慢的，并且与衰减相比消除更多偏置。

[0303] 移除偏置的第三种方法可以称为重配置。如上面那样，假设我们需要选择 12 个元素之一。我们注意到：使用 16 个叶子节点，我们具有 2:1 的偏置，其中 8 个节点 1 条道路可达，并且 4 个节点 2 条道路可达。假设我们建立三条道路来标识叶子节点到元素的映射。对于 12 个元素中的每一个，其出现在设置在两个配置中的“1 条道路”中并且在设置在一个配置中的“两条道路”中。我们然后(使用重选)来选择三个配置之一，并且然后选择使用 16 个叶子节点的数从而导致几乎完美的非偏置选择，并且我们仅仅必须在 3 个元素而非 12 个元素上应用重选。以很少的额外数据的代价，当消除偏置所需的配置数量小时，该方法提供紧致性和速度的最佳组合。(配置的最大可能数量通过从其进行选择的元素的数量被上限界，但是以该数量的配置，使用重配置是无意义的，因为选择配置仅仅是我们正尝试减小其偏置的问题的另一个实例。)

2.9 深非线性度：函数索引交织。在 [10] 中描述的 AES-128 实现使用 [21] 的方法来建立，已经使用 [4] 中攻击被渗透。尽管该攻击成功，但是该攻击很复杂，并且将需要大量人力来应用到任何特定软件实现，因此即使没有修改，[21] 的方法也是有用的。将极难对在根据 [22] 加强的根据 [21] 的实现上的攻击做出成功的 [4] 的攻击。然而，我们现在寻求更强的保护，并且因此使我们理应找到进一步保障 [21, 22] 的方法的方式以便致使诸如 [4] 中的那些的攻击不可行。

[0304] 2.9.1 浅非线性度和同态映射攻击。在根据宽输入线性实现（[21] 中的§4.0）和 pp. 9-10 上的§4.1（[21] 中的 [0195]-[0209] 段）中描述的矩阵分块方法的 [21, 22] 的实现中做出许多使用。以下是为真：[21] 的方法产生这样的线性变换矩阵的非线性经编码的实现。然而，实现是浅非线性的。即，这样的矩阵被转换成替换盒(查找表)的网络，其由

于空间限制而必然具有有限数量的元素。在用于对这样的盒编索引的值上以及在从这样的盒获取的元素值上的非线性编码(任意1对1函数,它们自身可表示为替换盒;即,作为查找表)由于空间限制而类似地限制为有限范围。

[0305] 因此,这样的经分块矩阵表示的输入输出编码的实现计算的任何数据变换直到I/O编码是线性的,所述分块矩阵表示被实现为替换盒的网络,或者用于表示本质上任意的随机函数的类似设备;即,通过对每个输入矢量元素单独重编码和对每个输出矢量元素单独重编码,可以将任何这样的变换转换成线性函数。

[0306] 在[4]中的攻击方法是基于同态映射的攻击类的实例。在$GF(2^8)$上的情况中,攻击利用线性函数的已知属性,因为这是AES中的计算的代数基础。具体地,在$GF(2^n)$中的加法使用按比特$\oplus$(异或)来执行,并且该函数定义精确已知形式的拉丁方。因此,有可能搜索从$\oplus$的经编码的查找表版本到未编码的版本的同态,并且有可能在任何函数$f = Q \circ \oplus \circ Q^{-1}$(其中$\oplus$是按比特的)的情况下以合理效率找到对特定仿射A的近似解$\hat{Q} = Q \circ A$(即,在实Q的仿射映射A内的近似$\hat{Q}$)。这些事实在[4]的攻击中被利用,并且存在可以类似地利用以下事实的其他攻击:[21, 22]的分块矩阵函数实现直到I/O编码是线性的。尽管这样的攻击仅产生部分信息,但是它们可以将针对精确信息的搜索缩窄到这样的点,在该点中,剩余的可能性可以通过穷尽搜索来考察。例如,使用[21, 22]提供的建立块的加密或解密的白盒实现可能易受诸如[4]中的攻击的密钥提取攻击或基于同态映射的相关攻击的攻击。

[0307] 2.9.2 挫败同态映射:深度非线性函数。对同态映射攻击的解决方案是用其为(1)宽输入的函数替代这样的矩阵函数;即,包括单个输入的比特数大,使得可能的输入值的集合极大,以及(2)深度非线性的;即,可能不可能通过I/O编码(即,通过单独对单独输入和单独输出重编码)被转换成线性函数的函数。

[0308] 使输入宽使得通过在所有输入上对函数制表来进行的暴力求逆消耗不切实际的巨量存储器,并且深非线性度防止同态映射攻击,诸如[4]中的攻击。

[0309] 例如,我们可以用深非线性MDS变换来替代AES中的MixColumns和InvMixColumns变换,从而致使这些中的任一个的暴力求逆不可能,AES中的MixColumns和InvMixColumns变换输入和输出32比特(4字节)值,所述深非线性MDS变换输入和输出64比特(8字节)值。称这些变型为$MixColumns_{64}$和$InvMixColumns_{64}$。(因为对消息的加密在发送方完成并且解密在接收方完成,所以这些在正常情况下将不再相同网络节点上呈现,所以攻击者在正常情况下只具有对它们之一的访问。)

例如,假设我们希望在$GF(2^n)$上(其中n是用于实现的多项式——即比特串——大小)或相应地在$\mathbb{Z}/(2^n)$上(其中n是期望元素宽度)构造这样的深非线性矢量对矢量函数。令u+v=n,其中u和v是正的非零整数。令G=我们选择的$GF(2^n)$的(相应地$\mathbb{Z}/(2^n)$的)表示,G_u=我们选择的$GF(2^u)$的(相应地,$\mathbb{Z}/(2^u)$的)表示,并且G_v=我们选择的$GF(2^v)$(相应地,$\mathbb{Z}/(2^v)$)的表示。

[0310] 假设我们需要实现深非线性函数$f: G^p \rightarrow G^q$,$p \geq 3$且$q \geq 2$;即,在我们选择的

$GF(2^n)$的表示 G 上的 p 矢量到 q 矢量的一个映射。

[0311] 如果我们想要线性函数，则我们可以使用 G 上的$q \times p$矩阵来构造线性函数，并且如果我们想要非线性但是直到1/o编码是线性的函数，则我们可以使用根据 [21, 22] 的这样的矩阵的分块经编码实现。然而，这些方法不足以获得深线性度。

[0312] 我们注意到G, G_u, G_v的元素都是比特串（长度分别为n, u, v）。例如，如果 n=8 并且 u=v=4，则 G 的元素是 8 比特字节，并且G_u和G_v的元素是 4 比特半字节（半个字节）。

[0313] 以下构造称为函数索引交织。我们引入运算$\mathbf{extract}[r,s](\cdot)$和$\mathbf{interleave}(\cdot,\cdot)$，它们在实际的任何现代计算机上是可容易实现的，如对于通过编译器在代码生成中创造的那些将明白的。对于比特串$S=(b_0,b_1,\ldots,b_t)$，我们定义$\mathbf{extract}[r,s](S)=(b_r,b_{r+1},\ldots,b_s)$，即，$\mathbf{extract}[r,s]$向 s（包含性的）返回比特 r。对于比特串的矢量$V=(S_1,S_2,\ldots,S_z)$，我们定义

$$\mathbf{extract}[r,s](V)=(\mathbf{extract}[r,s](S_1),\mathbf{extract}[r,s](S_2),\ldots,\mathbf{extract}[r,s](S_z))$$

，即$\mathbf{extract}[r,s]$向旧矢量元素的每一个的 s（包含性的）返回包含比特 r 的新矢量。对于相同长度的比特串的两个矢量，即$V=(S_1,\ldots,S_z)$和$W=(T_1,\ldots,T_z)$，我们定义$\mathbf{interleave}(V,W)=(S_1\|T_1,\ S_2\|T_2,\ \ldots,\ S_z\|T_z)$；即，$\mathbf{interleave}(V,W)$的每个元素是 V 的对应元素与 W 的对应元素的级联。

[0314] 为了获得上述我们的深非线性函数$f:\ G^p\mapsto G^q$，我们按照图 8 的流程图如下继续。

[0315] 80 选择函数$L:\ G_u^p\mapsto G_u^q$，或者可替代地选择G_u上的$q\times p$矩阵。（因为奇异平方子矩阵可能产生对于同态映射的脆弱性，所以优选地，L 的矩阵表示的最大平方子矩阵是非奇异的。如果 L 是 MDS，则没有 L 的平方子矩阵是奇异的，所以该优选无疑是令人满意的。）

82 选择$k\geq 2$函数$R_i:\ G_v^p\mapsto G_v^q,\ i=0,\ldots,k-1$，或者可替代地，选择$G_v$上的$k\geq 2$ $q\times p$矩阵。（因为奇异平方子矩阵可能产生对于同态映射的脆弱性，所以优选地，$R_0,\ldots,R_{k-1}$的矩阵表示的最大平方子矩阵是非奇异的。如果$R_0,\ldots,R_{k-1}$是 MDS，则没有任何R_i的平方子矩阵是奇异的，所以该优选无疑是令人满意的。）

84 选择函数$s:\ G_u^p\mapsto\{0,1,\ldots,k-1\}$，对于该函数，$s\{G_u^p\}=\{0,1,\ldots,k-1\}$（即，选择为“映射到……之上”或“满射”的 s）。

[0316] 与 s 是映射到……之上的要求不同，我们可以随机选择 s。然而，即使简单的构造也足以获得 s。作为示例，我们给出对于 s 的优选构造如下。

[0317] 如果 $k \leq u$，则我们选择线性函数 $s_1 : G_u^p \mapsto G_u$（或者等同地，在 G_u 上的 $1 \times p$ 矩阵）以及函数 $s_2 : G_u \mapsto \{0, 1, \ldots, k-1\}$。类似地，如果 $u < k \leq 2u$，我们可以选择线性函数 $s_1 : G_u^p \mapsto G_u^2$ 和函数 $s_2 : G_u^2 \mapsto \{0, 1, \ldots, k-1\}$，等等。于是，令 $s = s_2 \circ s_1$。在优选实施例中，k 是 2、1 或 8，或 2 的一些其他幂。

[0318] 假设 k=2。于是，s_2 可以返回 G_u 的元素的比特串表示的低阶比特；如果 k=4，则 s_2 可以返回低阶 2 比特，并且一般地，如果 $k \leq u$，则 s_2 可以返回比特串模 k 的值，对于 $k = 2^m$ 的我们的优选选择，其也就是说通过提取 s_1 输出的 m 低阶比特来获得。

[0319] 上面的优选方法使得我们使用针对 s_1 的分块矩阵实现，使得 [21, 22] 的方法应用于其。此外，当 f 是可逆时，我们可以通过下述方法使用该优选构造来直接获得 f^{-1} 的实现，这生成其构造类似于 f 的构造的 f^{-1} 函数。对于任何 $V \in G^p$，令

$$V_u = \mathbf{extract}[0, u-1](V)\ ,\ \text{以及}$$
$$V_v = \mathbf{extract}[u, n-1](V)\ ,$$
$$f(V) = \mathbf{interleave}(\ L(V_u),\ R_j(V_v)\ )\ ,\ \text{其中}\ j = s(V_u)\ .$$

[0320] 上面的步骤（4）中定义的函数 f 可以是或不可以是深非线性的。然后，接下来的步骤是检查深非线性度。我们使用以下测试来确定此。

[0321] 如果 f 是深度非线性的，则如果我们冻结除了至常数值的一个输入以外所有其输入，并且忽略除了一个输出以外的所有其输出，则我们获得 1×1 射影 f'。如果我们针对冻结的输入选择不同的值，则我们可以获得不同的 f' 函数。对于线性函数或者直到 1/O 编码是线性的函数，通过针对冻结的输入选择不同的值而可获得的独特 f' 函数的数量是容易计算的。例如，如果 $p = q$ 且 f 是 1 对 1（即，如果 $L, R_0, \ldots, R_{k-1}$ 是 1 对 1），则确切地存在 $|G|$ 这样的函数，如果 $q \geq p$，则 f 在该构造中仅可以是 1 对 1。

[0322] 我们简单地对这样的 f' 函数计数，其表示为在 G 上的 $|G|$ 矢量（例如，通过使用哈希表来存储当 p-1 个冻结的输入常数在所有可能性上改变时每个矢量的出现次数）。如果独特 f' 函数的数量不能通过用 $p \times q$ 矩阵替代 f 来获得，则 f 是深度非线性的。

[0323] 注意到我们可以不在 f 上而是在 f 的任意 1×3 射影上执行以上测试，我们可以加速该测试，其中 g 是通过冻结除了至常数值的三个输入以外的所有输入并忽略除了一

个输出以外的所有输出来获得的。这减少了针对从$|G|^{p-1}$到$|G|^2$的给定未冻结输入和给定未忽略输出要计数的函数实例的数量，这可以提供相当大的加速。此外，如果f是深度非线性的，则我们通常发现这在测试期间相当快：第一次我们发现射影函数计数不可从矩阵获得，我们直到g是深度非线性的，并因此f是深度非线性的。

[0324] 如果我们通过使用具有随机选择的三个输入和一个输出的g来使用假设并且我们在证明f的深非线性度时未成功，则f可能直到$1/O$编码是线性的。

[0325] (注意，射影实例计数有可能是可通过矩阵获得的，但是f仍然是深度非线性的。然而，这不可能偶然发生并且我们可以忽略它。在任何情况中，如果上述测试指示f是深度非线性的，则其无疑是深度非线性的。即，在针对深非线性度的测试中，以上测试可以生成误否定，但是决不会误肯定)。

[0326] 如果在步骤 88 中的测试未表明f是深度非线性的(或者对于紧跟该列表的变量，是充分深度非线性的)，则我们返回到步骤 80 并再次尝试。

[0327] 否则，我们终止该构造，已经获得期望的深度非线性函数f。

[0328] 作为以上的变型，我们可能希望获得是深度非线性的函数f，并且不仅如此，而且其射影也是深度非线性的。在该情况中，在以上步骤 88 中，我们可以增加具有随机选择的三个输入和一个输出的独特组的g函数的数量，对此，我们必须示出f'实例计数是不可通过矩阵获得的。我们测试的这些越多，我们越确保f不仅是深度非线性的，而且在其值域的所有部分上都是深度非线性的。我们必须平衡这样的测试的成本与获得深度非线性函数的重要性，其被保证在越来越多的其值域上是深度非线性的。

[0329] 2.9.3 实验验证。用于构造深度非线性函数f的方法优选实施例的 1000 此伪随机试验以伪随机生成的 MDS 矩阵 L 和$R_0, R_1\ (k=2)$来尝试，其中$f: G^3 \mapsto G^3, G = \mathrm{GF}(2^8)$并且$G_u = G_t = \mathrm{GF}(2^4)$。MDS 矩阵是使用具有伪随机选择的独特系数的范德蒙德矩阵方法来生成的。产生的 1000 个函数中有 804 个是深度非线性的；即，在 804 次执行构造方法中，步骤 88 指示该方法在其首次尝试中产生了深度非线性函数。

[0330] 类似的实验被执行，其中取代使用根据优选实施例的选择器函数$s = s_2 \circ s_1$的是，函数s_2被实现为 16 个 1 比特元素的表，每个元素是从集合$\{0,1\}$中伪随机选择的。1000 个这样的函数中有 784 个是深度非线性的；即，在 784 此构造中，步骤 88 指示构造方法的首次尝试产生了深度非线性函数。

[0331] 最后，类似的实验被执行，其中 s 被创建为从G_u^3映射到伪随机选择的$\{0,1\}$的元素的表。在 1000 次伪随机试验中，这产生 997 个深度非线性的函数。然而，其需要可设

置大小的表(512 字节用于该小实验,并且 2048 字节用于类似的函数 $f: G^4 \mapsto G^4$,具有与 AES 的 MixColumns 矩阵相同的 $1/O$ 维度)以存储 s。

[0332] 于是,我们看到上面给出的用于在有限域和环上创建深度非线性函数的构造方法——以及具体地,其优选实施例——是很有效的。此外,创建生成的深度非线性函数的逆是直接的,如我们将在下面看到的。

[0333] 2.9.4 以上构造的属性。如上所述那样构造的函数 $f: G^p \mapsto G^q$ 具有以下属性:

(1) 如果 L 和 $R_1, \ldots, R_k$ 是 1 对 1 的,则 f 是 1 对 1 的;

(2) 如果 L 和 $R_1, \ldots, R_k$ 是双射的(即,如果它们是 1 对 1 和映射在……之上的,使得 $p = q$),则 f 是双射的;以及

(3) 如果 L 和 $R_1, \ldots, R_k$ 都是最大距离可分的(MDS;参见下文),则 f 是 MDS。

[0334] 在两个 k 矢量之间的汉明距离——也就是 $u = (u_1, \ldots, u_k)$ 和 $v = (v_1, \ldots, v_k)$——是在其处 u 和 v 不同的元素位置的数量,即,其是 $\Delta(u, v) = |\{i \in \mathbf{N} \mid i \leq k \text{ and } u_i \neq v_i\}|$。最大距离可分(MDS)函数 $f: S^p \mapsto S^q$(其中,S 是有限集合并且 $|S| \geq 2$)是这样的函数,对于该函数,对于任何 $x, y \in S^p$,如果 $\Delta(x, y) = d > 0$,则 $\Delta(f(x), f(y)) \geq q - d + 1$。如果 $p = q$,则这样的 MDS 函数始终是双射。通过冻结 $m < p$ 的至常数值的输入并忽略除了 $n < q$ 的输出之外的所有输出($n \geq 1$,使得 $f': S^m \mapsto S^n$)而获得的 MDS 函数 $f: S^p \mapsto S^q$ 的任何射影 f' 也是 MDS 函数。如果 S 是有限域或有限环并且 f 是通过 $q \times p$ 矩阵(MDS 矩阵,因为矢量变换其,计算是 MDS)计算的函数——也就是说 M,则通过删除除了 M 的 z 行之外的所有行并然后删除除了 z 列之外的所有列(其中 $z \geq 1$)而获得的任何 $z \times z$ 矩阵 M' 是非奇异的;即,M 的每个平方子矩阵是非奇异的。

[0335] 这样的 MDS 函数在密码学中是重要的:它们用于执行一种“理想混合”。例如,AES 密码 [16] 在其除了最后一个以外的每轮中采用 MDS 函数作为两个状态元素混合函数之一。

[0336] 2.9.5 对构造的函数求逆。当我们针对一些有限域或有限环 G 采用 1 对 1 (通常是深度非线性的)函数 $f: G^p \mapsto G^q$,我们常常也需要 f 的逆或至少相对逆。(根据 [21, 22],对应的情形是我们具有 1 对 1 线性函数 $f: G^p \mapsto G^q$,其在 $1/O$ 编码之后将是浅非线性的,我们需要它的逆或相对逆。然而,我们可以通过代之以使用深度非线性函数和(相对)逆来显著地增强 [21, 22]。)

我们现在给出方法,借助于该方法,这样的逆(如果 $p = q$)或相对逆(如果 $p < q$)针

对根据我们的方法创建的 1 对 1 f 函数(深度非线性或相反)被获得。

[0337] 对于任何双射函数 $f: S^n \mapsto S^n$，存在唯一函数 $f^{-1}: S^n \mapsto S^n :- f \circ f^{-1} = f^{-1} \circ f = \mathrm{id}_{S^n}$。如果 $f: S^m \mapsto S^n$ 且 $m < n$，则 f 不能是双射的。然而，f 仍可以是 1 对 1 的，在该情况中，存在唯一的相对逆 $f^{-1}: f\{S^n\} \mapsto S^m :- f^{-1} \circ f = \mathrm{id}_{S^m}$。即，如果我们忽略不能通过调用 f 产生的 S^n 中的矢量，则 f^{-1} 像针对可以通过调用 f 产生的矢量的逆那样进行动作。

[0338] 我们现在公开了用于只要 L 和所有 $R_0, \ldots, R_{k-1}$ 是 1 对 1 时(在该情况中 $q \geq p$)构造针对我们构造的函数 f 的这样的相对逆的方法。如果 $p = q$，则 L 和所有 $R_0, \ldots, R_{k-1}$ 是双射，并且 f 的这样的相对逆也是 f 的(普通)逆。

[0339] 该方法可以当函数 s(参见图 8 的步骤 84)从线性函数 s_1 构造并且最终函数 s_2 用于将 s_1 的输出映射到 $\{0, \ldots, k-1\}$ 上时采用该方法，其中 s_2 被计算为来自用 k 除以 s_1 结果的余数。(如果 k 是 2 的幂，则我们可以通过取 s_1 结果的 $\log_2 k$ 低阶比特来计算 s_2，其是方便的，但是实际上对于我们当前目的是不需要的)。

[0340] 我们定义线性函数 L^{-1} 和 $R_0^{-1}, \ldots, R_{k-1}^{-1}$ 分别为 L 和 $R_0, \ldots, R_{k-1}$ 的相对逆。(因为这些函数是通过矩阵计算的，所以它们的相对逆可以通过以下方式容易且有效地获得：通过高斯消元等——即通过在有限域和有限环上的线性代数领域中熟知的方法来同时求解线性等式。)

我们具有来自 f 的构造的 $s = s_2 \circ s_1$。我们定义 $s_1' = s_1 \circ L^{-1}$，其中 L^{-1} 是 L 的相对逆。(因此，通过在有限域和有限环上的线性代数领域中熟知的方法容易发现的 G_u 上的 $1 \times q$ 矩阵来计算 s_1'。)我们定义 $s' = s_2 \circ s_1'$。我们现在具有映成函数 $s': G_u^q \mapsto \{0, \ldots, k-1\}$。

[0341] 期望的相对逆——或如果 $p = q$ 的普通逆——是如下定义的函数 $f^{-1}: G^q \mapsto G^p$。

[0342] 对于任何 $W \in G^q$，令

$$W_u = \mathbf{extract}[0, u-1](W),$$
$$W_v = \mathbf{extract}[u, n-1](W),$$

以及

$$f^{-1}(W) = \mathbf{interleave}(\, L^{-1}(W_u),\ R_j^{-1}(W_v)\,)$$

其中 $j = s'(W_u)$。

[0343] 当$p = q$时，这仅仅是f的普通逆。当$p < q$时，函数仅对于$f\{G^p\} \subsetneq G^q$中的矢量表现得像逆。

[0344] 如果我们具有用于 s 的不受限形式，即如果不以上文的优选实施例那样来构造，则我们可以仍然对双射或 1 对 1 f求逆或相对求逆。例如，如果s简单地是在G_u^p的元素上的表，则如果我们定义新的表$s' = s \circ L^{-1}$，则以上针对f^{-1}的方程(但是使用该不同的s')保持正确。该新表s'可以通过以下方式来获得：遍历G_u^p的所有元素 e、确定 L(e)，以及用 s 的元素 e 的内容来填充在s'的元素 L(e) 元素中。

[0345] 2.10 静态单赋值(SSA)形式。假设我们用诸如 C 的语言编写程序。数据将包括标量变量、阵列和结构。

[0346] 例程(C 术语中的函数)典型地充当 MF。每个例程可以由控制流图(CFG：参见§2.1.3)表示，其中节点表示基本块(BB；即，在控制转移中实现的直线代码段)。

[0347] 如果且仅如果每个标量变量确切地具有一个确定赋值，则例程是相对于其标量变量的静态单赋值(SSA)形式。否则，其是静态多赋值(SMA)形式。

[0348] 我们注意到，一般不可能将任意 C 代码例程转换成相对于其在 C 语言本身内的标量变量的 SSA 形式。原因在于：在代码中可能存在这样的位置，该位置诸如在既具有 then 替代又具有 else 替代的 if 构造之后，其中两个不同数据流路径合并——即其中变量(也就是说 x)在 then 路径和 else 路径二者中都被赋值。类似的问题关于循环出现，所述循环可以从顶部进入或者从底部重新进入。

[0349] 为了处理该问题，添加特殊的赋值形式：ϕ-赋值。例如，在对 x 的 then 路径和 else 路径赋值的情况中，我们可以分别对赋值的变量 x_1 和 x_2 重命名，并且然后，紧接在那些路径在 if 构造的底部的合并之后，插入 ϕ-赋值 $x_3 = \phi(x_1, x_2)$。利用ϕ-赋值的外延，现在有可能将任意 C 例程转换成 SSA 形式，只要每个变量在起始时被初始化即可。(如果不是，则下面提到的进一步精炼足以使得该方法是完全一般的。)使用 [14] 中的转换，我们不需要利用以ϕ-赋值(以及ω-赋值；见下文)外延的 C 执行转换，而相反可以在进一步外延中执行它，其中变量可以如下所示地被加下标，使得对应的原始变量可以始终通过移除下标来找到。

[0350] 2.10.1 从 SMA 形式到 SSA 形式的转换。到 SSA 形式的转换涉及：

(1) CFG 中的支配者的计算，最好使用 [13] 中的支配算法来实现；

(2) 针对 CFG 中的赋值的支配前沿的计算，其确定针对ϕ-赋值的最优位置，最好使用 [13] 中的支配前沿算法来实现；以及

(3) 代码到 SSA 形式的转换，最好使用 [14] 中的算法来实现(减去其支配者和支配前沿部分，其通过来自 [13] 的算法来替代)，注意：本论文中的转换方法会失败，除非原始 C 例程中的每个变量都在支配所有其他使用的位置中被定义(即，被赋值了一值)。如果这不是用于任何变量 v 的情况，我们可以通过在例程开始时、紧跟在 ENTER 指令之后、在例程从 SMA 到 SSA 形式的转换之前添加该形式 $v = \omega$ 的初始ω-赋值(表示对未定义值的初始化)来解

决它。

[0351] 2.10.2 从 SSA 形式到 SMA 形式的转换。从 SSA 形式到 SMA 形式的逆向转换是试验。

[0352] (1)每个ϕ-赋值取来自各种基本块(BB)的其输入,其可以容易在 SSA 形式中被标识,因为仅存在一个静态位置,在该静态位置处,任何标量变量被设置为值。当不同于ω-赋值的指令的输出计算作为至ϕ-赋值的输入的变量时,紧接在将指令的输出拷贝到作为ϕ-赋值的输出的变量的指令之后插入 MOVE 指令。

[0353] (2)移除所有ϕ-赋值和ω-赋值。

[0354] 注意,逆向转换不恢复原始程序。然而,其产生 SMA (即,可执行)形式的语义等同程序。即,如果我们将原始例程(当用诸如 C 的语言编写时始终采用 SMA 形式)转换成 SSA 形式并然后转换成 SMA 形式,则程序的最终 SMA 形式不可能等同于原始 SMA,但是其在功能上等同于原始 SMA。

[0355] 3. 基函数对和它们的实现

在本节中,我们提出用于生成白盒暗门单向函数其特别是用于生成在互逆对中的基函数双射的方法,使得被提供有这样的对中的成员的实现的攻击者不能容易地找到用于其逆(所述对的另一个成员)的实现,该攻击者也不能容易地找到用于提供的实现的点逆。

[0356] 3.1 白盒暗门单向函数:定义。我们定义一般什么是暗门单向函数,然后处理白盒情况,以及以在实现中将熵分离成密钥和随机化熵结束。

[0357] 3.1.1 暗门单向函数。我们从取自 [31] 的以下定义开始。

[0358] 总计$f: X \mapsto Y$是单向函数,$\textbf{iff}\ \forall x \in X\, f(x)$是“容易”计算的,但是对于“几乎所有”$y \in f\{X\}$计算$x :- f(x) = y$是“计算上不可行的”。

[0359] 以上f是暗门单向函数,$\textbf{iff}\ f$是单向函数,并且给定 s,对于任何$y \in f\{X\}$要找到$x :- f(x) = y$是计算上不可行的。(如果f是双射,则f^{-1}是这样的 s)。

[0360] 3.1.2 白盒暗门单向函数。对于来自 [31] 的上面的标准定义,我们为白盒攻击上下文添加以下非标准定义。

[0361] 以上f是白盒单向函数,$\textbf{iff}\ f$被设计为可实现的以便具有在白盒攻击下的单向函数属性。类似地,f是白盒暗门单向函数,$\textbf{iff}\ f$被设计为可实现的以便具有在白盒攻击下的暗门单向函数属性。

[0362] 双向$f: X \mapsto Y$是单向双射,$\textbf{iff}\ \forall x \in X\ f(x)$是“容易”计算的,但是对于“几乎所有”$y \in Y$,找到$x :- x = f^{-1}(y)$是“计算上不可行的”。

[0363] 以上双射f是暗门单向双射,$\textbf{iff}\ f: X \mapsto Y$是单向双射并且$\exists s :-$给定 s,对于任何$y \in Y$,找到$x :- x = f^{-1}(y)$是计算上不可行的。(例如,已经相对于密钥 s 进行了部分评估的对称密码f是暗门单向双射:秘密信息是密钥 s。)

以上双射 f 是白盒单向双射，$\mathbf{iff}$ f 被设计为可实现的，以便具有在白盒攻击下的单向双射属性。类似地，f 是白盒暗门单向双射，$\mathbf{iff}$ f 被设计为可实现的，以便具有在白盒攻击下的暗门单向双射属性。(例如，对称密码 f 的有效白盒固定密钥实现是白盒单向暗门双射：秘密信息 s 是密钥。)

N.B.：白盒暗门单向双射实现是双射 f 的实现，使得给定 f 的实现，在没有秘密信息的情况下，难以找到 f^{-1} 的计算上可行的实现。(特定的秘密信息可以本身是 f^{-1} 的计算上可行的实现。)

3.1.3 密钥熵和随机化熵。我们找到两个函数 f_K, f_K^{-1}，使得给定特定的构造算法，仅给定 K 可以找到 f_K 和 f_K^{-1} 的定义。K 是密钥熵。

[0364] 我们然后找到两种实现 $\mathbf{p}_{R_1}$ 和 $\mathbf{q}_{R_2}$，其中 $\mathbf{p}_{R_1}$ 实现 f_K，且 $\mathbf{q}_{R_2}$ 实现 f_K^{-1}。R_1 和 R_2 提供随机化熵，其不影响功能性；其仅用于确定在希望充分模糊实现以防止在一段时间内实现的破裂时如何表示该功能性。

[0365] $\mathbf{p}_{R_1}$ 和 $\mathbf{q}_{R_2}$ 包括我们的互逆基函数实现对。

[0366] 3.2 安全性：历史、理论和提出的方法。我们在互逆对中建立白盒暗门单向函数实现的初始尝试基于高阶多项式编码的歧义的难处理性的预期(参见 §2.3)、线性函数和深度非线性函数索引的交织的歧义的难处理性的预期(参见 §2.9.2)、线性函数加上在计算机算法上(即在 $\mathbf{Z}/(2^w)$ 上，典型的 $w = 32$) 用于消除这样的实现的 T 函数属性的一些其他操作的歧义的难处理性的预期。

[0367] 在结论上已经证明了该预期是错误的：单独借助于编码来产生难处理的歧义问题的任何这样的尝试失败，因为对于使用足够简单的编码对于有效地分析也是足够简单的。

[0368] 这导致我们寻求编程防卫，其中编码实际上扮演了一部分，但是其中它们与动态编程机制(控制流、例程调用、较宽数据组织等) 合作工作。与程序相关的许多问题是困难的。例如，我们已经证明对于在存在编码的情况下的控制流变平的冗余性和可达性问题是最坏情况 PSPACE 困难的 [9]。

[0369] 更一般地，我们具有 Rice 定理，其声明：对于部分函数的任何非平凡属性，不存在一般且有效的方法来决定给定算法是否计算具有该属性的函数。(“平凡”意思是属性或者适用所有部分函数或者不适用部分函数。) 该定理以 Henry Gordon Rice 命名，并且也以 Rice, John Myhill 和 Norman Shapiro 称为“Rice-Myhill-Shapiro 定理”。

[0370] Rice 定理的可替代声明如下。令 S 是非平凡的语言[2] 的集合，意味着：

(1) 存在认识 S 中的语言的图灵机(TM)，以及

(2) 存在认识不在 S 中的语言的 TM。

[0371] 于是不可决定任意 TM 决定的语言是否在 S 中。

[0372] Rice 定理仅应用于语言学属性而非操作属性。例如，可决定 TM 是否停止在给定输

入$\mathrm{in} \leq k$的步骤上，可决定 TM 是否停止在每个输入$\mathrm{in} \leq k$步骤上，以及可决定 TM 是否永远停止在$\mathrm{in} \leq k$步骤中。然而，病毒识别的一般不可能性是语言学的，并且 Rice 定律暗示完美地一般病毒识别器是不可能的。

[0373] Irdeto 的专利选集中的专利包括借助于以下实现的软件混淆和防篡改：数据流编码 [5、7、8]（标量和矢量及它们上的操作的编码）、控制流编码 [G]（对程序中的控制流的修改以使得其与来自计算的函数和代码块的多对多映射输入相关以计算它们）以及海量数据编码 [20]（基于软件的虚拟存储器或这样的存储器，在所述存储器中，逻辑地址被物理散射且还通过后台处理被动态重编码和物理地随时间而环绕移动）。

[0374] 在以上用于混淆软件并使其防篡改的方法中，数据流编码主要是静态过程（但是变量相关的编码使其潜在地是稍微动态的，在所述变量相关编码中，用于对某些变量和矢量的编码的系数由其他变量和矢量提供），而控制流编码和海量数据编码主要是动态的：数据结构被静态规划，但是对这些软件保护的实际操作很大程度上是在运行时对这些数据结构执行的动态操作的结果。

[0375] 控制流编码主要目的是（1）减小挫败动态攻击的职责，以及（2）通过将程序的正常控制流埋在相当大的外部控制流中来保护控制流的歧义。海量数据编码原始时的目的是找到将在存在高度动态别名的情况中正确工作的编码：例如，在 C 程序中使得积极地使用指针。

[0376] 以上形式的动态编码的困难在于支持数据结构（用于控制流编码的分派和寄存器映射表，用于海量数据编码的虚拟存储器编 / 解码表和地址映射表）本身泄漏信息。

[0377] 我们提出通过移动它们从运行时到编译时支持的大多数专门处理来使用类似于控制流编码和海量数据编码的组合的保护，但是在专门数据结构中具有大幅度减少。让我们称该新形式的保护为动态数据识别编码，该形式的保护具有控制流编码和海量数据编码的很多动态可变性，但是具有从运行时到编译时移动的专门数据结构。

[0378] [2] 语言是在字母表上的串的集合。字母表是有限的非空集合。

[0379] 在运行时消除大多数专门数据结构的益处在于，支持动态数据识别编码的操作变得更难将其与其应用于的代码进行区分。

[0380] 3.3 选择$\mathbf{Z}/(2^w)$上的可逆矩阵。为了创建在$\mathbf{Z}/(2^w)$上的可逆矩阵 M 及其逆 M^{-1}，我们如下继续。

[0381] 选择在对角线上和之上具有非零元素的上三角可逆矩阵，其中$n \times n$上三角可逆矩阵$U = [u_{i,j}]_{n \times n}$被选择使得：

- 如果 i<j，$u_{i,j} \leftarrow 1 + \mathbf{rand}(2^w - 1)$，
- 如果 i=j，$u_{i,j} \leftarrow 1 + 2 \times \mathbf{rand}(2^{w-1})$，以及
- 如果 i>j，$u_{i,j} \leftarrow 0$。

[0382] 因为所有对角元素是奇数，所以 U 无疑是可逆的。

[0383] 独立地选择两个这样的随机上三角矩阵 X，Y。于是$M \leftarrow XY^{\mathrm{T}}$，并且$M^{-1} \leftarrow (Y^{\mathrm{T}})^{-1}X^{-1}$。

[0384] 该方法确保逆的计算非常容易，因为所有逆在上三角矩阵上被计算，所述上三角矩阵已经为行阶梯形式，其中所有前导行元素是$\mathbb{Z}/(2^w)$的单位。

[0385] 3.4 虚拟机器和指令集。提出的实现形式是编程的和操作的。我们因此在虚拟机器方面以单个控制线程(没有并行性、没有时间切片)从底部对其进行定义，所述控制线程具有基于如用于 Intel IA32 和 Intel IA64 指令集架构二者的现代编译器(用于 GNU/Linux 的 gcc，用于 MS Windows 的 CL.EXE)所实现的 C 语言操作的指令集，其中(带符号)整型或无符号整型的默认大小是 32 比特。VM 指令集在没有上溢检查的情况下操作，在除非有另外声明的情况下将 32 比特字解释为无符号量。

[0386] 3.4.1 根指令。根指令包括七个字段：29 比特操作码、三个 1 比特文字标志 L1、L2、L3 以及三个 32 比特操作数(操作数 1、操作数 2、操作数 3)，如果对应的文字标志被设置，则它们中的每一个是文字的并且否则是寄存器号(参见图 2)。所有 VM 指令使用该格式，除了 ENTER 和 EXIT 以外，ENTER 和 EXIT 使用图 3 中示出的格式。以 24 比特字段保持计数 k、5 比特操作码、都设置为 0 的三个 1 比特文字标志，以及 k 个 32 比特寄存器号。

[0387] 在表 3 中示出了根指令集。

[0388] 3.4.2 隐含指令和基本指令集。在表 1 中示出隐含指令，每个基于基本指令的特殊使用。

[0389] 包括根和隐含指令的集合包括 VM 的基本指令集。

[0390] 3.4.3 与 C 语言运算符的对应。基本指令紧密对应于 C 语言中的运算符。

[0391] 布尔比较 EQ、NE、ULT、ULE、SLT、SLE 以及它们的隐含对应方 UGE、UGT、SGT、SGE 如果比较为真产生 1 并且否则产生 0。所有算法是没有上溢检查的 32 比特算法，因为是典型的 C 语言实现，并且所有算法是无符号的，除非在另外指出的情况下。

[0392] 注 意：ADD、SUB、MUL、DIV、REM、AND、OR、XOR、EQ、NE、ULT、ULE、UGT、UGE、LLSH 以及 LRSH 分别对应于 C 二进制运算符 `+, -, *, /, %, &, |, ^, ==, !=, <, <=, >, >=, <<, >>`，其中无符号整型操作数假设 32 比特无符号整型。类似地，NEG、NOT 分别对应于 C 运算符`-, ~`，具有 32 比特无符号整型操作数。最后，SLT、SLE、SGT、SGE 分别对应于具有 32 比特整型操作数的`<, <=, >, >=`，`llsh, arsh` 分别对应于具有 32 比特整型操作数和正移位计数的 C `<<, >>`，并且 LOAD、STORE、JUMP、JUMPZ、JUMPNZ 的能力对应于省略比特字段、函数指针和函数调用的 C 的存储器访问和控制能力。

[0393] 32 比特整型和无符号整型上的以上表现不是 ISO/IEC C 标准的部分，但是其是由用于 Intel IA32 与 IA64 的 GNU Linux 上的 gcc 和 Windows 上的 CL.EXE 提供的默认表现。

[0394] 因此，在以上 VM 指令集语义与用于 C 实现的实际标准之间存在紧密对应。

[0395] 3.4.4 宏指令。宏指令代表在基函数对的实现中使用的习惯用法。每个宏指令可以非常容易扩展到仅包含基函数的代码体中(参见 p.34 上的§3.4.2)，可能具有对一些额外临时寄存器的使用。合理的扩展是明显的并因此被省略。

[0396] 用于宏指令的一般形式粗略为助记［参数］ $d_1,\ldots,d_n \leftarrow s_1,\ldots,s_m$，表示取包括寄存器值 $(s_1,\ldots,s_m)$ 的输入矢量并产生包括寄存器值 $(d_1,\ldots,d_n)$ 的输出矢量的

映射。

[0397] （一个或多个）参数可以是值，或者一个或多个源寄存器（典型地，仅一个：s_0），或者一个或多个指令的序列，在该情况中，内部序列通过其的序列是参数的外部宏指令来修改。

操作数	助记符	操作数	效果
0	HALT	（被忽略）	停止执行
1	ADD	$d_1 \leftarrow i_2, i_3$	$\mathbf{Z}/(2^{32})$ 中的 $d_1 \leftarrow i_2 + i_3$
2	SUB	$d_1 \leftarrow i_2, i_3$	$\mathbf{Z}/(2^{32})$ 中的 $d_1 \leftarrow i_2 - i_3$
3	MUL	$d_1 \leftarrow i_2, i_3$	$\mathbf{Z}/(2^{32})$ 中的 $d_1 \leftarrow i_2 \times i_3$
4	DIV	$d_1 \leftarrow i_2, i_3$	$\mathbf{N}_0$ 中的 $d_1 \leftarrow \lfloor i_2/i_3 \rfloor$
5	REM	$d_1 \leftarrow i_2, i_3$	$\mathbf{N}_0$ 中的 $d_1 \leftarrow i_2$ 模 i_3
6	AND	$d_1 \leftarrow i_2, i_3$	按比特的 $d_1 \leftarrow i_2 \wedge i_3$
7	OR	$d_1 \leftarrow i_2, i_3$	按比特的 $d_1 \leftarrow i_2 \vee i_3$
8	XOR	$d_1 \leftarrow i_2, i_3$	按比特的 $d_1 \leftarrow i_2 \oplus i_3$
9	EQ	$d_1 \leftarrow i_2, i_3$	$d_1 \leftarrow [i_1 = i_2]$
10	NE	$d_1 \leftarrow i_2, i_3$	$d_1 \leftarrow [i_1 \neq i_2]$
11	ULT	$d_1 \leftarrow i_2, i_3$	$d_1 \leftarrow$ [$i_1 < i_2$ 无符号]
12	ULE	$d_1 \leftarrow i_2, i_3$	$d_1 \leftarrow$ [$i_1 \leq i_2$ 无符号]
13	SLT	$d_1 \leftarrow i_2, i_3$	$d_1 \leftarrow$ [$i_1 < i_2$ 带符号]
14	SLE	$d_1 \leftarrow i_2, i_3$	$d_1 \leftarrow$ [$i_1 \leq i_2$ 带符号]
15	LLSH	$d_1 \leftarrow i_2, i_3$	$\mathbf{N}_0$ 中的 $d_1 \leftarrow (i_2 \times 2^{i_3})$ 模 2^{32}
16	LRSH	$d_1 \leftarrow i_2, i_3$	$\mathbf{N}_0$ 中的 $d_1 \leftarrow \lfloor i_2/2^{i_3} \rfloor$
17	ARSH	$d_1 \leftarrow i_2, i_3$	$\mathbf{N}_0$ 中的 $d_1 \leftarrow \lfloor i_2/2^{i_3} \rfloor$ ；则设置（i_3 模 32）
18	LOAD	$d_1 \leftarrow i_2, i_3$	$d_1 \leftarrow M$ [$\mathbf{N}_0$ 中的 $(i_2 + i_3)$ 模 2^{32}]
19	STORE	$s_1 \rightarrow i_2, i_3$	M[$\mathbf{N}_0$ 中的 $(i_2 + i_3)$ 模 2^{32}] $\leftarrow s_1$
20	JUMPZ	$i_1?\ i_2, i_3$	如果 $i_1 = 0$，则 $\mathbf{Z}/(2^{32})$ 中的 $\mathrm{PC} \leftarrow i_2 + i_3$
21	JUMPNZ	$i_1?\ i_2, i_3$	如果 $i_1 \neq 0$，则 $\mathbf{Z}/(2^{32})$ 中的 $\mathrm{PC} \leftarrow i_2 + i_3$
22	JUMPSUB	$d \leftarrow i_2, i_3$	$d \leftarrow \mathrm{PC}+4;\ \mathrm{PC} \leftarrow i_2 + i_3$
23	ENTER	$\rightarrow d_1, \ldots, d_k$	进入例程，设置寄存器 $d_1, \ldots, d_k$ 为例程输入
24	EXIT	$\leftarrow s_1, \ldots, s_k$	退出例程，从寄存器 $s_1, \ldots, s_k$ 取例程输出

图例：

$s_1, \ldots, s_k$	源操作数：寄存器；不可以是文字
$d_1, \ldots, d_k$	目的操作数：寄存器；不可以是文字
i_1, i_2, i_3	输入操作数：寄存器内容或文字
$M[a]$	源 / 测试存储器位置，地址 =a
$x \leftarrow v$	用值 v 替代 x 的内容
PC	程序计数器（下一指令的地址）

表 3 虚拟机器根指令集。

[0398] 宏指令包括以下

$$\text{SAVEREGS}[\,]\ s_1,\ldots,s_n \to r$$

隐含指令 转译

MOVE $d_1 \leftarrow i_2$	OR $d_1 \leftarrow i_2, 0$
NEG $d_1 \leftarrow i_3$	SUB $d_1 \leftarrow 0, i_3$
NOT $d_1 \leftarrow i_2$	XOR $d_1 \leftarrow i_2$,（$\mathbb{Z}/(2^{32})$中的 -1）
UGT $d_1 \leftarrow i_3, i_2$	ULT $d_1 \leftarrow i_2, i_3$
UGE $d_1 \leftarrow i_3, i_2$	ULE $d_1 \leftarrow i_2, i_3$
SGT $d_1 \leftarrow i_3, i_2$	SLT $d_1 \leftarrow i_2, i_3$
SGE $d_1 \leftarrow i_3, i_2$	SLE $d_1 \leftarrow i_2, i_3$
JUMP i_2, i_3	JUMPZ 0? i_2, i_3

表 4 隐含虚拟机器指令。

[0399] 在$M[d+i-1]$中存储s_i，$i=1,\ldots,n$；然后设置$r \leftarrow r+n$。典型地，r将指定栈指针寄存器。

[0400] ---

$$\text{RESTOREREGS}[\,]\ d_1,\ldots,d_n \leftarrow r$$

设置$r \leftarrow r-n$；然后从$M[d+i-1]$加载d_i，$i=1,\ldots,n$。典型地，r将指定栈指针寄存器。

[0401] ---

$$\text{LINEARMAP}[{}^{n}_{m}M]\quad d_1,\ldots,d_n \leftarrow s_1,\ldots,s_m$$

其中${}^{n}_{m}M$是$\mathbb{Z}/(2^{32})$上的$m \times n$介质，在$s=(s_1,\ldots,s_m)$和$d=(d_1,\ldots,d_n)$的情况下，其计算$d=Ms$，d、s 在计算期间视为列矢量；

$$\text{SHORTROTATE}[\,]\quad d_1,\ldots,d_n \leftarrow i_0;\ s_1,\ldots,s_n$$

其中，使用寄存器或文字i_0并令

$k \leftarrow (-1)^{i_0 \bmod 2}(\lfloor i_0/2 \rfloor \wedge 31)$，

$\wedge$是在$\mathbb{B}^{32}$上按比特计算的，其当$k \geq 0$时在$\mathbb{Z}/(2^{32})$上计算

$$d_i \leftarrow 2^k s_i + \lfloor 2^{k-32} s_i \rfloor$$

$i=1,\ldots,n$，并且当$k<0$时计算

$$d_i \leftarrow 2^{32+k} s_i + \lfloor 2^k s_i \rfloor$$

$i=1,\ldots,n$；

LONGROTATE[] $d_1,\ldots,d_n \leftarrow i_0;\ s_1,\ldots,s_n$

其中，使用寄存器或文字 i_0 并令 N_0 上的

$$S=\sum_{i=1}^{n}2^{32(n-i)}s_i \quad 和 \quad D=\sum_{i=1}^{n}2^{32(n-i)}d_i$$

在 $\mathbf{Z}/(2^{32n})$ 上当 $k\geq 0$ 时计算

$$D\leftarrow 2^k S+\lfloor 2^{k-32n}S\rfloor$$

$i=1,\ldots,n$，并且当 $k<0$ 时计算 $D\leftarrow 2^{32n+k}S+\lfloor 2^k S\rfloor$

$i=1,\ldots,n$ ；

SEQUENCE[$i_1,\ldots,i_k$] $d_1,\ldots,d_n \leftarrow s_1,\ldots,s_m$

其中，$i_1,\ldots,i_k$ 是指令，其依次计算指令，其中 $s_1,\ldots,s_m$ 是序列从其输入的寄存器，并且 $d_1,\ldots,d_m$ 是其输出到的寄存器 ；

SPLIT[$c_1,\ldots,c_k$] $d_1,\ldots,d_k \leftarrow s_1$

其中对于 $i=1,\ldots,k$，$2\leq c_i\leq 2^{32}-1$ 并且 $1<\prod_{i=1}^{k}c_i\leq 2^{32}-1$，赋值

$$d_i\leftarrow\left\lfloor\frac{s_1}{\prod_{j=1}^{i-1}c_j}\right\rfloor \text{模}\ c_i$$

$i=1,\ldots,k$，其中我们假设 $\prod_{j=1}^{0}e(j)=1$，其中 e 是任意函数 ；即，根本没有元素的乘积始终是 1 ；

CHOOSE[$i_1,\ldots,i_k$] $d_1,\ldots,d_n \leftarrow s_0;\ s_1,\ldots,s_m$

其中 $i_1,\ldots,i_k$ 是指令，其计算单个指令 $i_{\lfloor 2^{-32}s_0k\rfloor+1}$，其中 $s_1,\ldots,s_m$ 都是通过 $i_1,\ldots,i_k$ 中的任一个用作输入的寄存器，$d_1,\ldots,d_m$ 都是通过 $i_1,\ldots,i_k$ 的任一个用作输出的寄存器，并且可以或不可以通过 CHOOSE 指令用作输入或输出的任何寄存器必须既表现为 $s_1,\ldots,s_m$ 中的输入，又表现为 $d_1,\ldots,d_m$ 中的输出 ；其中 CHOOSE 宏指令扩展到基本 VM 指令使用尽可能接近平衡的二进制比较树 ：例如，如果 $k=16$，则在平衡的二进制树中存在 15 个分支，每个分支具有形式

UGE $d\leftarrow s,n$
JUMPNZ $d?\ T$

其中 T 是当寄存器 $s\geq n$ 时控制转移到的标签，并且在树根处的 n 的值为 2^{31}（分割值的

上半部和下半部)，在其左子孙处为2^{30}(分割值的第一和第二四分之一 1)，以及在其右子孙处为$2^{31}+2^{30}$(分割值的第三和第四四分之一)，依此类推：

$$\text{REPEAT}[a,b,i] \quad d_1,\ldots,d_n \leftarrow i_0;\ s_1,\ldots,s_m$$

其中i_0是文字或寄存器，执行

$$\text{i} \quad d_1,\ldots,d_n \leftarrow s_1,\ldots,s_m$$

$a+\lfloor 2^{-32}(b-a+1)i_0 \rfloor$倍；

$$\text{PERMUTE}[p_1,\ldots,p_k] \quad d_1,\ldots,d_n \leftarrow s_0;\ s_1,\ldots,s_n$$

其中$p_1,\ldots,p_a$的每一个是$(1,\ldots,n)$的置换，其执行$d_i \leftarrow s_{p_i(i)}$，$i=1,\ldots,n$，其中$t \leftarrow \lfloor 2^{-32}s_0 k \rfloor + 1$(即，其在$s_1,\ldots,s_n$中放置通过$s_0$选择的$d_1,\ldots,d_n$的置换)，使用相同的尽可能接近平衡的二进制比较树和分支来扩展到基本 VM 指令，这已针对 CHOOSE 宏指令进行了描述。

[0402] ---

$$\text{RECODE}[e_1,\ldots,e_n] \quad d_1,\ldots,d_n \leftarrow s_1,\ldots,s_n$$

对于$i=1,\ldots,n$计算$d_i \leftarrow e_i(s_i)$，其可以显式地通过直接代码或者隐式地通过分式来实现；

$$\text{ENCODE}[e_1,\ldots,e_n] \quad d_1,\ldots,d_n \leftarrow s_1,\ldots,s_n$$

对于$i=1,\ldots,n$计算$d_i \leftarrow e_i(s_i)$，实现优选是e_i's的显式应用，而不是e_i's的隐式(即，分式)应用。

[0403] ---

$$\text{FRACTURE}[e_1,\ldots,e_n] \quad d_1,\ldots,d_n \leftarrow s_1,\ldots,s_n$$

对于$i=1,\ldots,n$计算$d_i \leftarrow e_i(s_i)$，实现优选是e_i's的隐式(即，分式)应用，而不是e_i's的显式应用；以及

$$\text{FUNIXINT}[m_{\text{left}},n_{\text{left}},m_{\text{right}},n_{\text{right}};\text{i}_{\text{left}},\text{i}_{\text{sel}},\text{i}_{\text{right}}] \quad d_1,\ldots,d_n \leftarrow s_1,\ldots,s_m$$

其中$m=m_{\text{left}}+m_{\text{right}}$并且$n=n_{\text{left}}+n_{\text{right}}$，计算

$$\text{i}_{\text{left}} \quad d_1,\ldots,d_{m_{\text{left}}} \leftarrow s_1,\ldots,s_{m_{\text{left}}}$$

以获得左结果，

$$\text{i}_{\text{sel}} \quad \sigma \leftarrow s_1,\ldots,s_{m_{\text{left}}}$$

以获得选择子σ，以及

$$i_{right}\quad d_{m_{left}+1},\ldots,d_n \leftarrow \sigma,\ s_{m_{left}+1},\ldots,s_m$$

通过函数索引交织的简化变量来获得获得右结果，该函数索引交织不一定是双射，但是明显地，只要对于$x \in$｛左，右｝$m_x = n_x$就是双射(即，总计、1 对 1 以及映射在……之上)，i_{left}计算双射函数，并且i_{right}的任何射影通过任意固定σ以计算双射函数来获得。

[0404] 3.5 建模程序结构。我们使用来自所谓的红龙书的模型 [1]。

[0405] 对于不同于分支指令 JUMP、JUMPZ 和 JUMPNZ 的每个基本 VM 指令，下一执行的指令是紧跟在当前指令后面的指令。因此，在控制流方面，程序形成其中仅在开始时进入并仅在结束时离开的直线代码块对应于图中的节点的图。

[0406] 因此，在构造期间，我们将我们的函数例程实现视为控制流图(CFG，参见§2.1.3)。每个节点用基本块(BB)来标识：其中第一指令是例程中的第一指令(始终是 ENTER 指令)或者是分支指令的目的地的 VM 指令序列，在基本块中没有其他指令是分支指令的目的地，并且基本块的最后指令始终是 EXIT 指令的分支。

[0407] 我们将控制流图限制为具有单个宿节点，并且仅该宿节点的 BB 在 EXIT 指令中结束，使得在例程中确切地存在一个 ENTER 指令和一个 EXIT 指令。

[0408] 在 CFG 的节点内，除了任何 ENTER、EXIT 或分支指令之外，指令可以被视为其中指令的定序仅受它们间的相关性限制的数据流图。这些具有三种：

(1)指令 y 与指令 x 是输入输出相关的，iff y 取作输入，一值通过 x 产生为输出，或者 y 与是与 x 输入输出相关的指令输入输出相关的。

[0409] (2)LOAD 指令 y 与 STORE 指令 x 是加载存储相关的，iff y 从一位置加载值，之前 x 在该位置中存储所述值，或者如果 y 与是与 x 加载存储相关的指令是加载存储相关的。

[0410] (3)STORE 指令 y 与 STORE 指令 x 是存储存储相关的，iff x 在 y 后续在其中进行存储的位置中进行存储，或者如果 y 与是与 x 存储存储相关的指令是存储存储相关的。

[0411] 执行相关性要求 y 在 x 之后执行。基本块的定序于是受约束于与其包含的指令的拓扑排序一致的定序，所述拓扑排序将这些相关性用作部分顺序。

[0412] 4. 标记 I：“WOODENMAX”命题

我们在这里引入用于在互逆对中建立白盒暗门单向函数的方法的我们的原始命题。

[0413] 该命题利用以下：

(1)置换多项式

模数$\geq 2^{32}$的$\mathbb{Z}/(2^w)$用作针对整数值、加法以及与$\mathbb{Z}/(2^w)$中的常数的乘法的编码。

[0414] 我们预期这是实际的，这归因于已知的限制：即$\mathbb{Z}/(2^w)$上的每个置换－多项式等同于次数$\leq w+1$之一。我们预期这样的编码具有高歧义性，这归因于对要相加或相乘以及编码要采用的数的非常大量的选择：每个置换－多聚具有许多等同的置换－多项式。

[0415] (2)§2.9 的函数和函数逆的构造，具有非常大量的右侧函数索引(即，针对选择子的大限制 n)。

[0416] 这将产生细粒度的互逆函数对的深非线性度，从而使得对基本矩阵的直接攻击更

加困难。

[0417] (3)在构造中的接口处的分式的使用(参见索引中的“编码分式”)——包括g_i间的分式——针对$i=1,\ldots,n$起作用(因为g_i不需要采用相同的编码,只要它们都是 1 对 1 的即可)。

[0418] 分式的效果是使构造的非线性度进一步变深。

[0419] (4)初始输入和最后输出按对混合以将产生自构造的深非线性度均匀地散布在输入和输出空间上,并且f输入和输出按对混合以及g_i输入和输出按对混合($i=1,\ldots,n$)以挫败同态映射攻击。

[0420] 在图 4 中在 p.40 上示出用于在互逆函数将 8 矢量映射到 8 矢量的情况构造,使得 BVM 的交织

文档关于函数索引的交织的定理及其反演结果采用 4 矢量到 4 矢量的f映射以及将 4 矢量映射到 4 矢量的g_i函数,$i=1,\ldots,n$。我们将在描述初始构造本身之后讨论最后调整。

[0421] 40 八个输入矢量元素通过 2x2 混合操作在对中混合,所述混合操作可以通过编码矩阵映射技术来执行,其中所述矩阵在诸如$\mathbf{Z}/(2^{32})$或$\mathbf{Z}/(2^{64})$的环上。矩阵操作被扩展到经编码的操作中,其中编码是基于环的置换多项式。仅三种操作需要被编码,只要混合器执行的映射是两个变量的线性或仿射加法(addvar)、常数与变量的线性或仿射加法,以及变量与常数的乘法(mulcon)。

[0422] 2x2 混合的目的在于此。如果函数索引的交织被朴素应用,则从输入 1、3、5、7 到输出 1、3、5、7 的射影直到 I/O 编码是线性的,但是从输入 2、4、6、8 到输出 2、4、6、8 的射影不是。通过混合输入 1、2、3、4、5、6、7、8 以及在(10)下面中混合输出 1、2、3、4、5、6、7、8,我们确保从输入到输出的这样的射影以及直到 I/O 编码的线性不存在。

[0423] 41 在以上步骤 40 之后,我们具有八个中间量,根据通过函数索引的交织选择子做出的选择,其中的 1、3、5、7 将被引导到左侧,且 2、4、6、8 将被引导到g_i函数之一。

[0424] 选择哪些右侧函数将被使用的选择函数通过 1x4 映射器、在环上的另一多项式编码的仿射操作来计算。该映射器精确地取与左侧函数相同的输入,以确保§2.9.5 的求逆方法工作。

[0425] 42 通过将中间量 2、4、6、8 从步骤 40 引导到合适的g_i实现,开关执行$g_{\sigma(\cdot)}$的实际选择。

[0426] 尽管这在图 4 中示出为中间量到选择的g_i实现的简单切换,但是实践中,操作可能远多于此。具体地,高度期望选择 n 的数量是极大的,使得我们的互双射白盒暗门单向函数实现展现出细粒度的深非线性度。为了实现此,我们必须在有限的空间中表示巨量的g_i实现。作为开始,考虑在$\mathbf{Z}/(2^w)$上的任何非奇异 4x4 矩阵,其中w预期是 32、64 或甚至更大。假设每行和列是唯一的,通过置换行和列,我们可以获得 576 个独特矩阵。我们还可以改变在边界处的编码(即,在其中在图 4 中箭头从一个块传递到另一个块的点处),并且许多其他改变方法存在。因此,我们可以通过合适地实现该“开关”而相当容易地在有限量空间

中获得大量g_i的提供。

[0427] 43 在图 4 的左侧，我们将输入混合到对中的f。这被进行以挫败同态映射攻击。在这样的攻击中，我们具有一种种类的运算$z = x \diamond y$，并且对于任何给定未编码的运算$\diamond$，存在三个间隙要填充。因此，可能猜测的数量通过r^3来上限界，其中 r 是环的元素的数量，即通过2^{3w}。现在，文档关于函数索引的交织的定理及其反演结果采用 4 矢量到 4 矢量的f映射以及将 4 矢量映射到 4 矢量的g_i函数，$i = 1, \ldots, n$。我们将在描述初始构造本身之后讨论最后调整。

[0428] 40 八个输入矢量元素通过 2x2 混合操作在对中混合，所述混合操作可以通过编码矩阵映射技术来执行，其中所述矩阵在诸如$Z/(2^{32})$或$Z/(2^{64})$的环上。矩阵操作被扩展到经编码的操作中，其中编码是基于环的置换多项式。仅三种操作需要被编码，只要混合器执行的映射是两个变量的线性或仿射加法(addvar)、常数与变量的线性或仿射加法，以及变量与常数的乘法(mulcon)。

[0429] 2x2 混合的目的在于此。如果函数索引的交织被朴素应用，则从输入 1、3、5、7 到输出 1、3、5、7 的射影直到 I/O 编码是线性的，但是从输入 2、4、6、8 到输出 2、4、6、8 的射影不是。通过混合输入 1、2、3、4、5、6、7、8 以及在(10)下面中混合输出 1、2、3、4、5、6、7、8，我们确保从输入到输出的这样的射影以及直到 I/O 编码的线性不存在。

[0430] 41 在以上步骤 40 之后，我们具有八个中间量，根据通过函数索引的交织选择子做出的选择，其中的 1、3、5、7 将被引导到左侧，且 2、4、6、8 将被引导到g_i函数之一。

[0431] 选择哪些右侧函数将被使用的选择函数通过 1x4 映射器、在环上的另一多项式编码的仿射操作来计算。该映射器精确地取与左侧函数相同的输入，以确保$g_i g_i^{-1}$的求逆方法工作。

[0432] 42 通过将中间量 2、4、6、8 从步骤 40 引导到合适的g_i实现，开关执行$g_{\sigma(i)}$的实际选择。

[0433] 尽管这在图 4 中示出为中间量到选择的g_i实现的简单切换，但是实践中，操作可能远多于此。具体地，高度期望选择 n 的数量是极大的，使得我们的互双射白盒暗门单向函数实现展现出细粒度的深非线性度。为了实现此，我们必须在有限的空间中表示巨量的g_i实现。作为开始，考虑在$Z/(2^w)$上的任何非奇异 4x4 矩阵，其中w预期是 32、64 或甚至更大。假设每行和列是唯一的，通过置换行和列，我们可以获得 576 个独特矩阵。我们还可以改变在边界处的编码(即，在其中在图 4 中箭头从一个块传递到另一个块的点处)，并且许多其他改变方法存在。因此，我们可以通过合适地实现该“开关”而相当容易地在有限量空间中获得大量g_i的提供。

[0434] 43 在图 4 的左侧，我们将输入混合到对中的f。这被进行以挫败同态映射攻击。在这样的攻击中，我们具有一种种类的运算$z = x \diamond y$，并且对于任何给定未编码的运算$\diamond$，存在三个间隙要填充。因此，可能猜测的数量通过r^3来上限界，其中 r 是环的元素的数量，即通过2^{3w}。现在，假设攻击者采用最大预期“生日悖论”优点。我们于是预期$2^{3w/2}$的

攻击复杂度，其对于 w=32 是 2^{48} 次尝试并且对于 w=64 是 2^{96} 次尝试。

[0435] 现在假设我们可以混合对中的输入。现在存在五个间隙来进行猜测，并且以上攻击复杂度（同样授予攻击者最大预期“生日悖论”优点）是 $2^{5w/2}$ 的攻击复杂度，其对于 w=32 是 2^{80} 次尝试并且对于 w=64 是 2^{160} 次尝试。明显地，于是如果我们可以密不可分地混合输入（并且对称的，输出）由此我们在至 f 的条目上具有 2×2 混合。

[0436] 44 在图 4 的右侧，我们将输入混合到在对中的 g_i。这被进行以挫败如在以上（4）中所提到的同态映射攻击，并且具有相同的优点。

[0437] 45 在左侧的 4×4 映射器是多项式编码的线性或仿射变换。

[0438] 46 在右侧的 4×4 映射器是多项式编码的线性或仿射变换。

[0439] 47 f 的实现通过在对中混合输出而结束。部分地，这被进行以直接使同态映射攻击更加困难；部分地，出于上面在 42 中给出的原因，这被进行以确保在逆函数中在对中混合输入，因为 f^{-1} 实现涉及与图 4 中示出的构成相反的构成，使得输入成为输出并且反之亦然。

[0440] 48 类似地，每个 g_i 的实现（在 BVM 文档的关于函数索引的交织的定理方面）通过在对中混合输出而结束，既使得同态映射攻击代价更大又确保在每个 g_i^{-1} 中在对中混合输入，因为 g_i^{-1} 实现涉及与图 4 中示出的构成相反的构成，使得输入成为输出并且反之亦然。

[0441] 49 最后，在对中混合最后输出，这可能通过多项式编码的 2×2 矩阵操作来进行，正如在 40 中那样初始输入为 n 阶以确保不存在多个输入到多个输出的直到 I/O 是线性的射影，而相反，所有这样的射影是深度非线性的。

[0442] 最后的调整修改图 4 中出现的结构如下。

[0443] • 通过使用标识（包括 MBA 标识），边界（在图 4 中通过表示数据流的箭头表示的）是模糊的，以便将图中用块表示的计算与后继块中的计算混合。因为我们具有对在块中使用的编码的完全控制，所以我们被良好定位以实现这样的模糊。

[0444] • 我们文字利用在 84、项（3）中提到的分式来帮助确保实现在每个级别都是非线性的。

[0445] • 我们优化该实现以确保足够的性能。例如，当采用许多多项式时，针对输入的所有使用计算一次该输入的幂。

[0446] 标记 II

标记 II 命题与标记 I（参见 p.39 上的 §4）类似，因为其具有固定的内部结构，仅仅在基函数实现对间有系数变化。

[0447] 5.1 初始结构：选择 f_K 和 f_K^{-1}。初始程序不采用内部阵列。除了对输入的初始访问和对输出的最后交付以外（即，除了 VM 指令 ENTER 和 EXIT 的实现以外），在程序中的所有运算是标量的并且不利用 LOAD 和 STORE 指令，它们仅严格地被需要用于编索引的存储器访问。

[0448] 所有计算在 B^{32} 上运算并且从 B^{32} 产生结果，但是对于 ADD、SUB、MUL，这样的值被

解释为$\mathbf{Z}/(2^{32})$的元素，并且结果是针对该模的环的合适$+,-,\times$结果。即，C 计算机运算+, -, *, &, |, ^和~是适于在没有上溢检查的典型 32 比特机器实现中的无符号整型操作数的那些运算。

[0449] 我们在f_K的构造中以及f_K的指定中于是在确切具有相同结构(但是当前具有不同系数)的f_K^{-1}的仅一个指定的容许中消耗 K（密钥)熵。

[0450] f_K或f_K^{-1}的基本结构在图 5（第一半）和图 6（第二半）中示出，其中圆圈 a, b, c, d, e, f, g, h 表示换页连接符。

[0451] 5.1.1 数据流。沿着图 5 和 6 中的箭头的数据流始终是$\mathbf{B}^{32}$(32 比特字)的元素，从箭头开始行进到箭头头部。当箭头分成两个方向时，字被运送到两个箭头头部表示的点。

[0452] 5.1.2 不相关。类似部件的随机选择。对于其中图 5 和 6 中的两个部件具有相同标签的f_K指定，它们独立于彼此地被选择；实际上，每个部件被随机且独立地选择，不受任何其他部件的影响。对于f_K^{-1}指定，当然，选择所有部件以便指定通过f_K定义的f_K的函数逆，使得一旦选择了f_K指定，f_K^{-1}指定就被固定：不消耗进一步的 K 熵来构造它。

[0453] 5.1.3 $n\times n$对$n:n$。在图 5 和 6 中的一些部件标签使用$n\times n$标签，诸如$4x4$ L 或$8x8$置换，而其他使用$n:n$标签，诸如$8:8$重编码或$4:4$重编码。

[0454] $n\times n$指示部件具有 n 个输入和 n 个输出，并且执行部件可以将熵从输入移动到非对应的输出(如在通过矩阵来映射矢量中那样)。

[0455] $n:n$指示部件具有 n 个输入和 n 个输出，并且执行部件仅可以将熵从输入移动到对应的输出：即，实际上，这样的部件由 n 个并排的标量运算构成，使得$4:4$重编码采取四个标量值，并且在没有与任何其他标量值交互的情况下单独地对每一个标量值重编码。

[0456] 5.1.4 选择部件。在图 5 和 6 中的选择部件被标记为选择 2 个$4x4$置换的 1 个或选择 2 个$4:4$重编码的 1 个。

[0457] 标记为选择 2 个$4x4$置换的 1 个的那些具有以下形式

$$\mathrm{PERMUTE}[p_1,p_2]\quad d_1,d_2,d_3,d_4 \leftarrow s_0 : s_1,s_2,s_3,s_4$$

其中p_1,p_2是随机选择的4×4置换，s_0经由来自$4x4S$。

[0458] 标记为选择 2 个$4:4$重编码的 1 个的那些具有以下形式

$$\mathrm{CHOOSE}[r_1,r_2]\ d_1,d_2,d_3,d_4 \leftarrow s_0 : s_1,s_2,s_3,s_4$$

s_0经由$4:4$重编码来自$4x4S$，其中每个r_i是以下形式的 VM 宏指令

$$\mathrm{RECODE}[e_1,e_2,e_3,e_4]\quad d_1,d_2,d_3,d_4 \leftarrow s_1,s_2,s_3,s_4$$

在重编码中的所有e_i编码是针对f_K随机选择的。

[0459] 5.1.5 函数索引的交织的四次出现。§2.9.2 中描述的函数索引的交织在f_K或f_K^{-1}指定中出现四次。每次其包括三个4×4线性映射(对于一些4×4矩阵 M，VM 宏指令

LINEARMAP[$\frac{1}{2}M$]……），在图 5 和 6 中分别标记为 4x4 L、4x4 S 和 4x4 R，它们的 4 × 4 矩阵是使用 §3.3 中的方法独立选择的：连同一个 4:4 重编码（VM 宏指令 RECODE[e_1, e_2, e_3, e_4]……四个 e_i 编码从可用的置换多项式编码随机选择），选择 2 个 4x4 置换的 1 个的两次出现，以及选择 2 个 4:4 重编码的 1 个的两次出现（参见 §5.1.4）。

[0460] 函数索引的交织的每个实例具有单个左侧函数和 $2^4 = 16$ 个右侧函数。

[0461] 5.1.6 其他部件。剩下的部件不在函数索引的交织的实例内，包括 8:8 重编码的三次出现，每次出现的形式为：

$$\text{RECODE}[e_1,\ldots,e_8] \quad d_1,\ldots,d_8 \leftarrow s_1,\ldots,s_8$$

以及 8X8 置换的两次出现，每次出现的形式为：

$$\text{PERMUTE}[p] \quad d_1,\ldots,d_8 \leftarrow 0 : s_1,\ldots,s_8$$

（对于 f_K）具有单个、随机选择的置换，以及 2x2 混合器的八次出现，每次出现的形式为：

$$\text{LINEARMAP}[\tfrac{2}{2}M] \quad d_1,d_2 \leftarrow s_1,s_2$$

其中 M 是通过 §3.3 中的方法针对 f_K 选择的 2×2 矩阵。

[0462] 5.2 混淆 f_K 或 f_K^{-1} 实现。以下方法用来模糊程序实现 f_K 或 f_K^{-1}，其中实现具有在 p. 43 上的 §5.1 中详述并在图 5 和图 6 中图示的通用结构。

[0463] 以下节中的变换被一个接一个地执行，除了在这些节的主体中另外提及的情况之外。

[0464] 5.2.1 拷贝删节。用于 f_K 或 f_K^{-1} 实现的朴素代码包含许多 MOVE 指令。当值通过中间寄存器经由 MOVE 从一个寄存器转移到另一个时，经常有可能消除中间步骤并将结果直接转移到最后目的地。

[0465] 这在置换的情况中特别有效，置换是使用一系列 MOVE 而朴素实现的。例如，对初始和最后 8x8 置换的简化意味着：随机选择的置换仅意味着哪个数据流源是第一个接收特定输入的数据流源是随机选择的，并且哪个数据流宿是最后交付特定输出的数据流宿是随机选择的。

[0466] 规则是可以通过普通级别的优化消除的任何 MOVE 必须被消除；即，混淆的 f_K 或 f_K^{-1} 实现的最后版本必须包含使用普通（即，非英勇）级别的优化可实现的最小数量的 MOVE 指令。

[0467] 更具体地，假设我们可以容易以 SSA 形式将值产生者与其值消耗者相关联，必须被省略的那些 MOVE 是可以通过对输出重新编号并重新转换成 SSA 直到没有进一步的拷贝删节发生来在 SSA 中移除的那些。

[0468] 当以下情况时 MOVE 可以被省略：该 MOVE 形成其中多个 MOVE 形成弧的运算树中的弧，原始值产生者（可能是 ϕ-赋值）是根，该根支配 MOVE 弧和消耗者，并且没有消耗者本身是 ϕ-赋值。

[0469] 拷贝删节可以在以下过程中的各种点处执行，并且无疑被完成为用于移除冗余 MOVE 指令的最后步骤。

[0470] 5.2.2 分支到分支删节。相关的删节形式可以对分支执行。如果基本块(BB)仅包含非条件分支指令(即，非条件 JUMP 基本指令)，则分支到该 BB 的任何分支的目标可以被修改为非条件分支。这可以重复，直到没有这样的分支到分支仍出现为止，并且仅包含非条件 JUMP 的任何不可达 BB 任何可以被移除。

[0471] 分支到分支删节可以在以下过程中的任何点处被执行，并且无疑被完成为用于消除分支到非条件分支序列的最后步骤。

[0472] 5.2.3 未用代码消除。当代码为 SSA 形式时，如果任何寄存器是指令 x 的输出，但是决不是指令 y 的输入，则指令 x 是未用代码。我们可以重复地移除所有这样的指令，直到没有未用代码保留为止。

[0473] 这可以在混淆期间的各种时间完成，并且无疑被完成为用于消除未用代码的最后步骤。

[0474] 5.2.4 独特动态值的＊哈希插入和生成。选择在 $\mathbf{Z}/(2^{w})$ 上随机选择的独特奇元素的 1×8 哈希矩阵，并且生成代码，所述代码通过该矩阵来映射原始输入，从而产生单个输出。放置该代码以用于紧跟初始 ENTER 指令的哈希矩阵计算(初始地，LINEARMAP 宏指令)。所述单个输出是输入的“哈希”，并且将用于生成用于存储器混洗的独特动态值 $C_1, C_2, C_3, \ldots$ (参见 §5.2.13)。

[0475] 接着，选择置换多项式(PP) P，并且其中 z 是包含来自以上矩阵计算的输出的输出寄存器，插入代码以生成在 $\mathbf{Z}/(2^{n})$ 上的 N 值 $C_i = P(z+i)$，其中 $n = \lceil \log_2 N \rceil$ 并且 N 的导数将在随后进行描述。初始地，PP 计算被插入为重编码宏指令，其中所有的编码等同地是 P。

[0476] 5.2.5 宏指令扩展。所有的宏指令被扩展为一系列基指令。该扩展是无关紧要的，并因此在此被省略。

[0477] 在扩展之后，仅基指令保留。

[0478] 5.2.6 ＊控制流复制。在 §5.2.4 中添加了一个矩阵，其将所有八个输出映射到一个输出，并且存在在图 5 和 6 中示出的 20 个矩阵，每个表示分别表示两个或四个输入到两个或四个输出的映射，21 个矩阵映射中的每一个扩展成没有内部分支(即，JUMP……)指令的基指令的连续序列 X。

[0479] 哈希矩阵计算初始时是第一矩阵映射计算。我们取代码用于要依次计算的每一“轮”中的水平行中的矩阵，诸如靠近图 5 和 6 中的 CB 结构的开始和结束的 $2x2$ 混合器行，或者 $4x4$ L、$4x4$ S 序列；即，首先是用于最左边的矩阵的代码，然后是用于在其右边的矩阵，依此类推。此外，在每一“轮”中，我们注意到用于计算 $4x4$ L 和 $4x4$ S 矩阵映射的计算的代码一定在用于计算选择 2 个 $4x4$ 置换的 1 个的代码之前，之后是选择 2 个 $4:4$ 重编码的 1 个，之后是 $4x4$ R 代码，之后是选择 2 个 $4x4$ 重编码的 1 个，之后是在图右侧的选择 2 个 $4x4$ 置换的 1 个。

[0480] 在§3.5中提到的表示方面，取决于我们正在处理 20 个矩阵中的哪一个，用于计算矩阵在其应用于的 2 或 4 矢量上的影响的代码在该表示中表现为直线代码序列，其：

（1）占据整个基本块(BB)，除了最后条件分支序列以外(条件分支跟随在之后的比较，采取 ULT、JUMPNZ 指令对的形式，紧跟矩阵计算；例如，这初始时是用于图 5 和 6 中的标记为4x4 R的情况，因为它们紧跟这样的点，在该点出，包含 then-BB 和 else-BB 的 if-then-else 结构选择如在图中由跟随 4x4 R的选择 2 个4:4重编码的 1 个所指示的两个重编码之一，所述 then-BB 和 else-BB 中的每一个以至 BB 的 JUMP 结束，所述 BB 包含用于4x4 R矩阵映射的代码，4x4 R矩阵映射之后是条件序列(ULT 之后是结束所述 BB 的 JUMPNZ)；或者

（2）在 BB 中出现，所述 BB 具有在所述 BB 的直线代码中在其之前及之后的计算指令；例如，这初始时是用于图 5 和 6 中标记为4x4 L和4x4 S的每一个矩阵的情况，以及用于在图 5 中标记为2x2混合器的所有矩阵和在图 6 中标记为2x2混合器的除最左侧矩阵以外的所有矩阵的情况；或者

（3）出现在 BB 的开始处并且之后是进一步的计算指令：这是用于在图 6 中的标记为2x2混合器的最左侧矩阵的情况；或者

（4）出现在 BB 的开始处并且之后是分支指令(JUMP……)或者条件分支指令序列(ULT、JUMPNZ)；这在开始时未在图 5 和 6 中出现，但是可能在以下描述的处理之后出现。

[0481] 在以下描述的方式中，我们用至代码块的两个拷贝之一的分支来替换每个这样的矩阵代码块，所述代码块的两个拷贝之一由至公共点的分支终止：即，实际上，用以下代码来替换代码 X：

$$\texttt{if } r < 2^{31} \texttt{ then } X_1 \texttt{ else } X_2$$

其中，r 是支配 X 的指令(并因此以上 if- 构造)的输出，并且支配指令是从可能的前驱随机一致地选择的。X_1, X_2分别是执行代码块 X 的 then 和 else 实例。

[0482] 为了实现以上变换，我们继续如下(按照图 9)：

900 如果实现当前不是 SSA 形式，则将其转换成 SSA 形式，如在§2.10.1中描述的。

[0483] 905 从具有单个输出并且支配用于主题矩阵映射的代码的第一指令的所有指令中随机一致地选择指令 I。

[0484] 910 将实现转换成 SMA 形式，如在 p.30 上的§2.10.2中所描述的。

[0485] 915 隔离包括用于主题矩阵的矩阵映射代码的连续指令序列。即，如果用于主题矩阵的第一矩阵映射代码未开始 BB，则在所述第一指令紧前放置非条件 JUMP 指令，从而在该点出将该 BB 一分为二，并且如果用于主题矩阵的最后矩阵映射指令未在 JUMP……或 EXIT 指令紧前，则在所述最后指令紧后插入非条件 JUMP 指令，从而在该点出将其 BB 一分为二。在该点处，原始 BB 已经被纵向切分成多个 BB 零次、一次或两次，并且用于主题矩阵映射的代码在其自己的仅包含该映射代码的 BB 中被隔离，所述映射代码之后是单个非条件 JUMP 指令。

[0486] 920 创建两个新 BB。第一 C（用于“选择”）仅包含 ULT、JUMPNZ 序列，用于实现

if $r < 2^{31}$ then ... else ... 决定。对于寄存器 r，我们使用以上我们选择的指令 I 的输出。令 X 是以上我们隔离的 BB，第二个是 X'，在每个方面中是 X 的精确拷贝，除了 X 是在原始作为控制流图（CFG）的图中的独特且初始隔离的图节点以外，但是当前其并不是，因为 CFG 不具有隔离的节点。

[0487] 925 如下用 C, X, X' 替换 CFG 中的原始 X。分别地，使得原始指向 X 的所有分支目标指向 C。在 $< 2^{31}$ 上使 C 的最终分支分支到 X，并且替代地在 $\geq 2^{31}$ 上分支到 X'。

[0488] 935 执行分支到分支删节（参见 §5.2.2）。

[0489] 注意，尽管这复制计算，但是其不产生用于比较的独特拷贝，因为在每次执行时，执行矩阵映射的两个路径中仅用于给定矩阵的路径被执行。

[0490] 5.2.7 来 - 自插入。如果实现当前不是 SMA 形式，则将其转换成 SMA 形式（参见 §2.10.2）。

[0491] 然后，对于计算 $r_H < 2^{31}$ 从而产生 1（如果为真）或 0（如果为假）并将其输入提供到 JUMPNZ（“如果为真则跳转”）指令的每个布尔比较 ULT 指令，随机选择两个常数 $c_1 \leftarrow \mathbf{rand}(2^{32})$ 和 $c_2 \leftarrow \mathbf{rand}(2^{32})$，并在比较 ULT 之后且在 JUMPNZ 指令取其输入之前插入代码，其中，所插入的代码计算 $r_e \leftarrow c_2 + (c_1 - c_2) r_H$。

[0492] 在 JUMPNZ 的真目的地处插入 $r_d \leftarrow c_1$，并且在 JUMPNZ 的假目的地处插入 $r_d \leftarrow c_2$。（回顾：当代码为 CFG 形式时，每个条件分支具有两个目标。）

记住，为了将来使用，计算 r_e 和 r_d 的指令的输出应当是相同的。

[0493] 5.2.8 * 数据流复制。在以下描述的方式中，对于不是 JUMP..., ENTER 或 EXIT 的每个指令，指令被拷贝（使得原始指令被其拷贝紧跟），并且对于所有拷贝的指令选择新寄存器，使得如果 x 和 y 是指令，y 是 x 的拷贝，则：

（1）如果 x 输入 ENTER 指令的输出，则对应的 y 输入使用相同的输出；

（2）如果 x 输入具有拷贝 v 的原始指令 u 的输出，则 y 的对应输入从 v 输入与 x 从其输入的 u 输出相对应的输出；以及

（3）如果 x 输出到 EXIT 指令，则 y 的对应输出输出到特殊的未用宿节点，以指示其输出被丢弃。

[0494] 因此，除了分支以外的所有计算具有原始和拷贝出现。

[0495] 为了实现该变换，我们如下继续（按照图 10）。

[0496] 我们添加新指令 JUMPA（“任意跳转”），其是在控制流图（CFG）形式中具有两个目的地的非条件分支，正如条件分支（参见 §3.5）那样，但是没有输入：替代地，JUMPA 指令在其两个目的地之间随机进行选择。JUMPA 实际上不是 VM 指令集的部分，并且没有 JUMPA 将在 f_K 或 f_K^{-1} 的最后混淆实现中出现。

[0497] 我们在以下变换过程中使用 JUMPA。

[0498] 1000 如果实现已经不是 SMA 形式，则将其转换成 SMA 形式（参见 §2.10.2）。

[0499] 1005 对于在实现 $X_1,\ldots,X_k$ 中的 BB 的每个 BB X_i，通过以下方式用三个 BB C_i,X_i,X_i' 替换其：创建等同于 X_i 的新 BB X_i'，并且添加仅包含目标为 X_i 和 X_i' 二者的单个 JUMPA 指令，使 X_i 和 X_i' 为 C_i 的 JUMPA 的两个目标，以及使指向 X_i 的每个非 JUMPA 分支目标替代地指向 C_i。

[0500] 1010 将实现转换成 SSA 形式(参见 §2.10.1)，隔离 X_i 和 X_i' 中的本地数据流，但是 X_i 和 X_i' 中的对应指令仍然计算相同的值。

[0501] 1015 将每个 X_i' 中的所有代码合并回其 X_i 中，在合并时使来自 X_i 和 X_i' 的指令交替，使得对应的指令对是相继的：首先是 X_i 指令，并且然后是对应的 X_i' 指令。

[0502] 1020 使为 C_i 的每个分支目标替代地指向 X_i，并且移除所有 C_i 和 X_i' BB。在该点处，数据流已经被复制，CFG 的原始形状已经被恢复，并且实现没有 JUMPA 指令。记住在每一个 X_i 中的指令对应以在将来使用。

[0503] 5.2.9 *随机交叉连接。如果实现当前不是 SSA 形式，则将代码转换成 SSA 形式(参见 §2.10.1)。归因于 SSA 形式的使用，已知为产生相同输出的以下一些指令 P_0 可以是 ϕ-赋值，从已知为产生相同输出的非 ϕ-赋值指令取它们的输入。

[0504] 归因于 §5.2.6、§5.2.8 和 §5.2.7 中应用的变换，许多指令属于静态已知为产生相同输出的对。这样的情况在这些节中已提到，具有添加的注解：关于这样的相同输出的信息应保留以供将来使用。我们现在利用该保存的信息。拷贝的数量始终为 2：两个用于数据流复制，两个用于导出的控制流复制(因为控制和数据流复制二者都已经应用于它们，但是控制流复制的计算的仅一个实例在执行所述实现中发生)，并且两个用于“来 - 自”插入。

[0505] 存在两种方式，其中实现中的 SSA 形式的指令对可以已知为具有与 §5.2.6、§5.2.8 和 §5.2.7 的动作的结果相同的结果。两个指令可以是彼此的数据流复制，或者两个 ϕ-赋值可以具有已知为彼此的数据流复制的输入，这归因于矩阵映射计算的控制流复制。

[0506] 令 u_1,u_2 是这样的对指令，其在这样的保存的信息的基础上已知为具有相同输出，每个 u_i 取 k 输入，其中如果指令是基指令(例如，NEG 或 MUL)，则 k 是一或二，并且对于跟随控制流复制的矩阵映射的 ϕ-赋值是二。

[0507] 以 $\frac{1}{2}$ 的概率，我们如下翻转对 u_1,u_2 输出的使用：对于消耗 u_1 输出(如果有的话)的每个指令，我们对其进行修改以替代地取 u_2 输出，并且反之亦然。

[0508] 我们对于每个可能的这样的对 u_1,u_2 重复此，直到没有这样的对仍要考虑进行翻转。

[0509] 该变换的效果如下。作为数据流复制的结果，除了实现的非常开始和结束以外，数据流被分割成决不重叠的两个独特的子图。在随机交叉连接之后，这两个数据流图已经被彻底合并成单个数据流图。

[0510] 5.2.10 *检查插入。如果实现当前不是 SSA 形式，则将其转换成 SSA 形式(参见

§2.10.1)。

[0511] 如在§5.2.9中的随机交叉连接那样，我们继续通过已知为具有与归因于§5.2.6、§5.2.8和§5.2.7中的处理的值复制的结果相同的输出的指令对，也即u_1, u_2。

[0512] 如在§5.2.9的，这样的指令可以是基指令或c-赋值，并且我们确切地使用相同的准则来如§5.2.9中那样标识这样的对。

[0513] 相继地选择具有归因于步骤§5.2.6、§5.2.8和§5.2.7产生的复制的已知相同输出的这样的指令对，直到每个对已经被选择为止。

[0514] 对于每个这样的指令对，也即u_a, u_b，从之前在这样的处理中未用作u_c的u_c的所有选择中随机一致地选择单个指令u_c，或者如果没有这样的选择存在，则从包括之前在这样的处理中用作u_c的那些的u_c的所有选择中选择单个指令u_c，使得u_c是由u_a, u_b二者支配的。(如果根本没有这样的u_c存在，则不进一步处理u_a, u_b对：简单地继续到下一对，或者如果没有保留，则根据本节终止处理。)

令o_a, o_b, o_c分别是u_a, u_b, u_c的输出。紧跟u_c地，放置代码以计算$o_d \leftarrow o_c + o_a - o_b$，并且使得$o_c$的所有输入者替代地输入$o_d$。(因为$o_a = o_b$，所以我们应当具有$o_d = o_c$，所以这应当没有净效应，除非攻击者篡改代码)。

[0515] 继续这样的处理，直到所有这样的对u_a, u_b已经被选择为止。

[0516] (结合§5.2.7的r_c, r_d检查帮助防止分支干扰；如果对中的一个成员被修改而没有修改另一个，则另一个通过使得下游计算出故障来帮助挫败扰乱攻击。)

5.2.11 代码转换。如果实现当前不是SMA形式，则将其转换成SMA形式(参见§2.10.2)。

[0517] 取每个二进制运算，其计算

$$z \leftarrow f(x, y)$$

其中，f是+、-、或*之一，并且用为以下的代数简化的计算来替换其

$$c_3(z) \leftarrow f(c_1^{-1}(x), c_2^{-1}(y))$$

或者等同地，用以下的代数简化来替换运算f

$$c_3 \circ f \circ [c_1^{-1}, c_2^{-1}]$$

使得对于将产生者连接到消耗者的每个弧，产生的值的编码(c_1函数)匹配消耗者假设的编码(其中，编码的逆被应用)。即，执行网络编码(参见§2.3.1)。

[0518] 以上输出z用作比较EQ、NE、ULT、UGT、ULE、UGE、SLT、SGT、SLE、SGE或条件分支JUMPZ或JUMPNZ的输入时，c_3必须是恒等式编码。此外，在原始程序中，作为重编码宏指令的最后输出得到的任何输出或者重编码宏指令的扩展不能被进一步修改；即，RECODE被取为进行明文计算，其的输出不能被编码。初始输入和最后输出也使用恒等式编码。即，其代码转换将改变程序计算的函数的任何输出被留下不被编码。

[0519] 在输入是常数 c 的情况下，用某个常数 $e_c(c)$ 替换它，并如同其来自于用编码 e_c 产生它的产生者一样对其进行处理。

[0520] 有时不可能使所有产生者和消费者编码在它们应当在的每个地方都匹配。在这发生的情况下，用输出编码 e_a 和输入编码 e_b 进行产生，并在弧上插入 $e_b \circ e_a^{-1}$ 以解决冲突。

[0521] 根据如在节 C 中所描述的针对该目的选择的方案，每个 e_i 是在 $\mathbb{Z}/(2^{32})$ 上的双射二次多项式(PP)函数，或者是这样的 PP 的逆。让我们简单地将它们称为 PPS。因为 PPS 仅涉及乘法和加法，所以 PPS 可以被计算为一系列仿射步骤，我们假设这为现在的情况。

[0522] 5.2.12 寄存器最小化。如果实现不是SMA形式，则将其转换成SMA形式(参见p.30上的§2.10.2)。

[0523] 我们现在得到针对寄存器中的生命跨度的冲突图。生命跨度在值被产生(即，被指令输出到寄存器)时开始并且在其中不做出对该值的进一步使用的点处(即，在上一次被指令输出的该值用作输入而在将该值放置在寄存器中并使用它之间对寄存器没有中间改变之后)结束：即，在被数据弧连接到消耗者的最后消耗者已经完成执行之后结束。如果两个生命跨度在不同产生者处开始并且在执行中存在二者在其处都已经开始并且它们中没有一个已经结束的点，则两个生命跨度冲突。

[0524] 我们可以将此视为其中生命跨度是节点并且如果且只有如果生命跨度冲突弧才连接两个节点的图。图的重要性在于，如果两个生命跨度冲突，则它们产生的值必须存储在不同寄存器中，而如果它们未冲突，则它们产生的值可以存储在相同寄存器中。

[0525] VM 允许不确定数量的寄存器(在任何速率——它们中的 2^{32} 个——良好)，但是我们的目的是使寄存器的数量最大化以通过潜在地使许多不同操作使用相同位置来增加混洗通过存储器的值的模糊性。

[0526] 从图中最小次数的节点开始，我们一次移除一个节点即它们的入射弧，直到所有节点已被移除为止。然后我们以相反顺序重新插入它们以及与它们一起被移除的任何弧，当我们重新插入它们时为它们选择寄存器。这是在 Chaitin 的算法上的变体，并且趋向于在独特颜色(寄存器)的数量趋向于朝最小数量的意义上产生有效的图着色(即，寄存器分配)。

[0527] 保留生命跨度信息和冲突图以供在 p.53 上的 §5.2.13 中进一步使用。

[0528] 5.2.13 *存储器混洗。如果实现不是SSA形式，则将其转换成SSA形式(参见p.29上的§2.10.1)。

[0529] 在实现中包括存储器阵列 A，其包含 2^p 个二进制字，其中 $p = \lceil \log_2 N \rceil$ (参见 p.47 上的 §5.2.4)。

[0530] 在代码中标识实现仿射映射 $y \leftarrow sx + b$ 或 $y \leftarrow s(x + \frac{b}{s})$ 的连续指令对(MUL，ADD)或(MUL，SUB)的所有实例(这取决于哪个指令在其他之前)，其中 s，b 和 $\frac{b}{s}$ 是常数输入并且 x 是非常数输入，并且其中或者 MUL 是输入 ADD 或 SUB 的输出的仅有指令或者反之亦然，并且其中仿射输出值 y 后续被一些其他指令用作输入。一旦这样的对已经被找到，就从进一步考虑中将其移除，但是继续直到所有这样的对已经被找出为止。称这些对为

$P_1,\ldots,P_N$。注意，每个这样的对仅具有单个非常数输入 x。

[0531] 与$P_1,\ldots,P_N$值$K_1,\ldots,K_N$相关联，这些值初始时全部等于 1。

[0532] 遍历$K_1,\ldots,K_N$。在每个K_i处，以$\frac{1}{2}$概率改变每个遍历的值至 2。遍历值为 2 的K_i，以$\frac{1}{2}$概率改变每个遍历的值至 3。遍历值为 3 的K_i，以$\frac{1}{2}$概率改变每个遍历的值至 4。

[0533] 在该点处，对于每个对P_i，存在值 K，具有属于集合$\{1,2,3,4\}$的值。

[0534] 定义$R_1,\ldots,R_N$如下。令R_i是在对的输出（y 由以上仿射映射计算）的生命范围中的在其处可以插入非分支基指令的点的数量，如果所述非分支基指令已被插入，则其将在生命范围中。

[0535] 定义$W_1,\ldots,W_N$如下。令W_i是重叠生命范围的最大指令集的基数（即，或者在生命范围中，或者通过输出其值开始生命范围，或者通过消耗其值来终止生命范围中的路径），使得集合中没有成员支配任何另一个成员。这估计生命范围中的“宽度”或路径重数。

[0536] 对于作为对P_i的输出的 y 的每个这样的生命范围，以作为 1 与K_iW_i/R_i中较小者的概率来选择在其处可以将指令插入如上提到的范围中的每个点处，使得在其中$W_i=1$的相当普通的情况中，在P_i的 y 输出的生命范围中存在预期数量K_i个这样的所选点。令针对给定生命范围所选点的集合为S_i，使得$|S_i|$的期望值 =K_i，其中$W_i=1$。（当然，$|S_i|$的实际值可能不同。）

如下定义$F_1,\ldots,F_N$。在对P_i的 y 输出的生命范围中，明显地，输入该 y 输出的每个指令（也即w）由产生 y 输出的对成员支配：将该对成员称为 m。如果对于每个这样的w，存在$s\in S$，使得 m 支配 s，s 又支配w，则$F_i=1$。否则$F_i=0$。

[0537] 我们现在将我们的注意力限制到对于其$F_i=1$的那些对P_i。对于每个这样的对P_i，我们在 A 中分配新集合$|S_i|+1$索引，$C_j,\ldots,C_{j+|S_i|}$（参见§5.2.4），使得P_i和S_i的每个成员具有其自己的所指派的索引。在针对给定P_i,S_i对的索引集合只有它们的对应P_i y 输出未由冲突图中的弧连接（即，如果它们的生命范围不重叠，参见§5.2.12）才可能与针对另一对的索引集合重叠的约束下，我们在P_i,S_i间尽可能多地重用索引。

[0538] 移除对P_i——（MUL，ADD）或（MUL，SUB）——用 RECODE、STORE、LOAD、RECODE 序列来替换其。每个 RECODE 将一个输入映射到一个输出，因此我们在每个存储和加载上对值重编码。于是，存在序列$s_1,\ldots,s_k,w$，使得以上最后的 RECODE 支配$s_1,\ldots,s_k,w$，其中$s_1,\ldots,s_k\in S_i$，w是输入被移除的P_i的 y 输出的指令，并且序列$s_1,\ldots,s_k,w$的每个元素支配其序列。作为结果，我们取被移除的序列的 x 输入，通过 2(k+1) 个 RECODE 来映射它，并且任何将它传递到w。我们修改最后 RECODE 使得一系列重编码的净效应是向 y 至w提供由w预期的输入编码，即，我们通过修改序列中的最后编码来引入计算$y\leftarrow sx+b$的分式。我们针对所有指令w重复此，一旦中间编码已被选择，就决不改变它（因为一些s_i可能出现在至多个 y 消耗者w的路径上）；即，如果已经针对一个路径选择了重编码，则不针对另

一重叠的路径而改变它们。

[0539] 我们如上针对对于其$F_i = 1$的每个对P_i而继续。我们然后将实现转换成 SMA 形式(参见§2.10.2),并且扩展所有重编码宏指令。

[0540] 5.2.14 随机指令重定序。如果实现不是 SSA 形式,则将其转换成 SSA 形式(参见§2.10.1)。

[0541] 确保冗余 MOVE 指令首先被省略(参见§5.2.1),如在§3.5 中使用基于相关性的部分定序对其指令的拓扑排序那样在每个 BB 中对指令重定序。在排序期间的指令的基数后继间,该后继被随机一致地选择。

[0542] 5.2.15 最后清除和代码删节。执行代码删节(参见§5.2.1),分支到分支删节(参见§5.2.2),以及未用代码消除(参见§5.2.3)。

[0543] 执行寄存器最小化(参见§5.2.12)以使寄存器(临时变量)的数量最小化,但是不尝试改变在混洗阵列 A (参见§5.2.13)中使用的位置的数量。当最小化完成时,代码为 SMA 形式。

[0544] 发射该代码。

[0545] 6. 透明盒标记 III

标记 III 命题不同于标记 I (参见§4)和标记 II (参见§5)的命题,因为其具有可变的内部结构,在该可变的内部结构中,系数和结构二者在基函数实现对间改变。

[0546] 如之前地,主要媒介是根据算法在互逆对中形成的像密码或哈希的函数实现的互逆对,属于非常大系列的这样的对,其中精确对通过算法和(典型地大)密钥 K 二者来确定。除了密钥信息 K 之外,形成对的算法消耗随机信息 R,其用于指定不影响对的外部行为而只影响内部处理的实现的那些混淆方面,通过该内部处理来实现该外部行为。

[0547] 6.1 设计原理。我们预期标记 III 用在这样的环境中,在该环境中,实现被曝露于白盒和 / 或灰盒攻击,并且在该环境中,利用标记 III 的应用的操作涉及跨网络的通信。

[0548] 6.1.1 安全性刷新速率。为了有效的应用安全生命周期管理,应用必须在持续基础上抵抗攻击。作为该抵抗的部分,我们预期这样的应用响应于包含安全更新信息的安全性刷新消息而自升级。这样的升级可以包括补丁文件、表替换、新密码密钥以及其他安全相关的信息。

[0549] 安全性的可行级别是这样的级别 :在应用中,安全性被足够频繁地刷新,使得损害实例的安全性要花费的时间比使损害无效的安全性刷新花费的时间更长 ;即,与实例可能典型地被破坏相比,它们被更快地刷新。

[0550] 这无疑可在非常高的安全性刷新速率处实现。然而,这样的频繁刷新动作消耗带宽,并且随着我们提高刷新速率,分配给安全性刷新消息的带宽比例增加,并且可用的非安全性有效载荷带宽降低。

[0551] 明显地,于是,设计合适的安全性刷新速率对于每种应用都是需要的,因为能容忍的开销取决于上下文有很大的变化。例如,如果我们预期在云应用中仅有灰盒攻击(相邻侧信道攻击),则与我们预期有白盒攻击(由恶意的云提供商职员进行的内部攻击)的情况下

相比，我们将使用更低的刷新速率。

[0552] 6.1.2 外部和内部脆弱性和攻击抵抗。假设我们的实现对实现函数f_K, f_K^{-1}，其中f_K, f_K^{-1}是 T 函数。则通过重复地应用这些函数中的任一个，我们可以精确地表征使用比特切片攻击的其计算。在这样的攻击中，我们首先这些函数的操作忽略了除低阶比特以外的所有比特，并且然后忽略低阶的两个比特，等等，从而获得信息，直到达到完整字大小（也即，32 比特）为止，在该点处，我们具有关于函数如何表现的完整信息，其等价于密钥 K 的知识。

[0553] 这是外部脆弱性。当攻击获得实现细节的知识时，其这样做而不用对实现那些细节的代码的进行任何检查，并且可以如对黑盒实现的自适应已知明文攻击那样被执行。

[0554] 如果所述对的函数具有以下属性，则存在较不严重的脆弱性：每个函数充当在特定域上的特定 T 函数，并且独特 T 函数的数量低。在该情况中，统计分桶攻击可以表征每个 T 函数。然后，如果域可以被类似地表征而也不用对代码进行任何检查，则使用自适应已知明文攻击，攻击者可以完整表征所述对的成员的功能性、完全绕过其保护，仅使用黑盒方法。

[0555] 明显地，我们必须确保独特 T 函数的有效数量足以挫败以上攻击。（在标记 III 实现中，每区段存在 10^8 以上个独特 T 函数，并且在整体上存在 10^{40} 以上个 T 函数。）

现在假设实现对包括实现全级联的函数（每个输出取决于每个输入，并且平均而言，改变一个输入比特会改变一半输出比特）。内部脆弱性的示例出现在标记 II 实现中，其中，通过在某点处“切割”实现，我们可以找到对应于矩阵的子实现（部件），使得相关性级别精确地是 2x2（在该情况中，部件是混合器矩阵）或 4x4（在该情况中，其是 L、S 或 R 矩阵之一）。一旦这些已经被隔离，则线性函数的属性允许这些矩阵的非常有效的表征。

[0556] 这是内部攻击，因为其需要非黑盒方法：其实际上需要对实现的内部进行检查，即是静态的（以确定相关性）还是动态的（以通过基于线性度的分析来表征矩阵）。

[0557] 作为一般性规则，我们可以更完全地挫败外部攻击，并且迫使攻击者进入不断增加的细粒度的内部攻击，攻击者的工作变得越难，并且最特别地，攻击者变得越难进行自动化。

[0558] 自动化的攻击是尤其危险的，因为它们可以有效地提供类破裂，这允许给定技术的所有实例被可以广泛分发的工具破坏。

[0559] 因此，我们寻求通过使用可变和不断变复杂的内部结构和不断更具变化的防御来创建这样的环境，在所述环境中

（1）实例的任何完全破裂需要许多子破裂；

（2）所需的子破裂在实例间不同；

（3）被攻击的部件的结构和数量在实例间不同；以及

（4）所采用的保护机制在实例间不同；

使得使攻击自动化变为足够大的任务，从而阻碍攻击者尝试它（因为在相当长时间内，时间将花费来建立这样的攻击工具，所部署的保护可以已经移动到新的技术，在所述新的技术中，攻击工具的算法不再够用）。

[0560] 6.2 初始结构：选择f_K和f_K^{-1}。标记 III 函数具有宽度为 12 个 32 比特字（总宽

度为 384 比特)的输入和输出。实现主要由一系列区段构成,其中每个区段是函数索引的交织(FII)的实例。该一系列区段之前和之后是意图挫败盲相关性分析攻击的初始和最后混合步骤(其在输入开始时执行,用于得到相关性关系,而不考虑攻击下的实现的结构细节)。

[0561] 我们将主要处理 f_K。鉴于我们熟知的 FII 方法,区段逆是相当明显的,并且通过以相反顺序串接 f_K 区段的逆发现总体 f_K^{-1} 以相反顺序夹在初始和最后混合步骤之间。

[0562] 每个这样的区段具有左函数、使用与左函数相同的输入的选择子计算以及右函数。右函数是 FII 的嵌套实例。因此,每个区段将输入和输出矢量划分成三个子矢量 :进入和退出外部左函数的部分、进入和退出内部左函数的部分,以及进入和退出内部右函数的部分。我们将把这些称为左、中和右子矢量。

[0563] 6.2.1 在 $\mathbb{Z}/(2^{32})$ 上选择矩阵。我们以两种不同方式选择矩阵 :

- 一般 :在没有元素是 0 或 1 并且所有元素是独特的约束下,在 $\mathbb{Z}/(2^{32})$ 上随机选择 $m \times n$ 矩阵 ;或者
- 可逆的 :根据在 p. 33 上的 §3.3 中给出的方法在 $\mathbb{Z}/(2^{32})$ 上选择 $n \times n$ 矩阵,但是具有额外约束 :产生的矩阵不包含具有值 0 或 1 的元素并且所有元素是独特的。

[0564] 6.2.2 * 初始和最后混合步骤。在 §2.8 中,我们给出用于使用具有排序网络拓扑的决定来置换元素序列或其他形式的选择的技术。通过用 2×2 双射矩阵替换条件交换来将每个输入混合到每个输出,我们可以精确地取得相同的网络拓扑,并且产生混合网络,所述混合网络初始时将 CB 函数的每个输入与彼此混合,并且我们可以在最后采用另一这样的网络来将 CB 函数的每个输出与彼此混合。如针对置换的情况那样,混合并不是完全均匀的,并且其偏置可以使用 §2.8.2 中的技术减小,但是再次地,条件交换被混合步骤替换。

[0565] 6.2.3 * 细分区段的输入和输出矢量。以下选择仅仅是示例 :具有不同宽度和更宽的划分大小的选择的许多其他选择是可能的。

[0566] 如果针对区段的输入子矢量以特定方式被静态划分,也即针对左、中和右分别的 3-5-4,任何其输出也被这样静态划分。

[0567] 对于以上允许的子矢量长度是三、四、五和六个 32 比特字。因为每个输入和输出矢量具有长度十二个 32 比特字(384 比特),所以其遵循从左至右然后向下的字典序的十个允许的配置为 :

3-3-6 3-4-5 3-5-4 3-6-3 4-3-5
4-4-4 4-5-3 5-3-4 5-4-3 6-3-3 。

[0568] 如果我们以从 0 到 9 的顺序对上面十个配置编号,则我们生成的针对每个区段所选的配置的号码静态地通过 rand(10) 来选择 ;即,我们在构造时随机一致地从以上十个可能性中进行选择。

[0569] 6.2.4 * 选择针对区段的输入和输出的定序。第一区段将初始输入输入到 f_K 或 f_K^{-1},并因此是输入无约束的。类似地,最后区段向 f_K 或 f_K^{-1} 进行输出,并因此是输出无约束的。因此,第一区段的输入或最后区段的输出分别随机一致地附着到初始输入或最后输出。

[0570] 在所有其他情况中,我们如下选择对区段的输入和输出的定序。

[0571] 我们注意到，对于任何区段，其左输出子矢量的输出仅取决于其左输入子矢量的输入，其中间输出子矢量的输出仅取决于其左和中间输入子矢量的输入，并且其右输出子矢量的输出取决于左、中和右子矢量的输入。

[0572] 我们因此静态地在以下约束下随机一致地将区段 Y 的输入链接到其在前区段 X 的输出。

[0573] (1)区段 X 的右输出矢量输出必须具有至区段 Y 的左输入矢量输入的最大数量的链接。例如，如果 X 是 6-3-3 区段并且 Y 是 3-4-5 区段，则 X 的右输出子矢量输出中的三个链接到 Y 的左输入子矢量输入中的三个。

[0574] (2)未在以上约束(1)下链接到区段 Y 的左输入矢量输入的任何区段 X 的右输出矢量输出必须链接到区段 Y 的中间输入矢量输入。例如，在以上 6-3-3 X、3-4-5 Y 的情况中，X 的右输出矢量输出中的剩余三个链接到 Y 的中间输入矢量输入中的三个。

[0575] (3)在满足以上约束(1)和(2)之后，区段 X 的中间输出矢量输出链接到 Y 的输入子矢量的最左可能输入子矢量，其中 Y 的左输入子矢量中的那些在最左边，Y 的中间输入子矢量中的那些在最左和最右之间的中间，并且 Y 的右输入子矢量中的那些在最右边。

[0576] 以上概述在于：当我们将信息从一个区段转移到下一个时我们静态地尝试使输入上的相关性最大化。我们始终被保证在两个区段之后实现“全级联”(使得每个输出取决于每个输入)，并且我们还尝试使携带这些相关性的数据流的宽度最大化(因此以上约束(2)和(3))。

[0577] 6.2.5 *区段的串接。f_K(并因此f_K^{-1})是区段的序列。每个区段的基本配置 r-s-t 根据 §6.2.3 被静态选择，并且每一个从原始输入或从其前驱静态地链接到 §6.2.4。构成 f_K(并因此f_K^{-1})实现的相继区段的数量从集合 {5,6,7} 被随机一致地选择。

[0578] 6.2.6 *非可变编码。在代码中的某些点处，我们使用非可变编码。非可变编码的意义——无论其是恒等式编码、线性编码还是置换多项式编码——在于当混淆应用于 f_K 或 f_K^{-1} 实现时其不能改变：其的存在是语义的部分并因此不能被修改。

[0579] 如果非可变编码被提及而没有对编码的指定，则置换多项式被采用。

[0580] 6.2.7 *创建区段。给定配置 r-s-t (例如)，我们如下创建 f_K 区段(按照图 11)：

1100 使用 §6.2.1 的可逆方法，选择 $r \times r$ 矩阵 L、$s \times s$ 矩阵 M 以及 $t \times t$ 矩阵 R，并且 24 个一致地随机选择的非可变编码：12 个应用于至这些矩阵的输入并且 12 个应用于它们的输出(将矩阵视为矢量映射函数)。让我们将具有它们的输入和输出非可变编码的这三个矩阵称为函数 $\mathcal{L}, \mathcal{M}, \mathcal{R}$。于是，$\mathcal{L} = L_{out} \circ L \circ L_{in}$、$\mathcal{M} = M_{out} \circ M \circ M_{in}$ 以及 $\mathcal{R} = R_{out} \circ R \circ R_{in}$ 附着到矩阵 X，$X \in \{L, M, R\}$，其中 X_{out} 执行非可变输出编码，并且 X_{in} 执行非可变输入编码。

[0581] 1105 使用 §6.2.1 的方法，选择具有对应的函数 C 的 $1 \times r$ 选择子矩阵 C，其取与 $\mathcal{L}$ 相同的输入并且具有输入编码 L_{in} 和输出编码 C_{out}；即，$\mathcal{C} = C_{out} \circ C \circ L_{in}$。(对应的 f_K^{-1} 区段

将具有形式为$C_{out} \circ C \circ L^{-1} \circ L_{out}^{-1}$的选择子——其当然将被简化。）

取 C 的输出的两个高阶比特并加 2 以形成范围从 2 到 5 中的迭代计数。比该迭代小一的迭代计数是范围从 1 到 4 中的数，其是在其输出被传递到后继区段之前，整个右侧函数的输出（取$s-t$个输入并产生$s-t$个输出）具有直接反馈到其输入的其输入并总体被再次执行的次数。

[0582] 1110 选择$2s$选择子函数$\mathcal{I}_1, \ldots, \mathcal{I}_s$和$\mathcal{O}_1, \ldots, \mathcal{O}_s$，每个类似于上面的 C。这些中的高阶四个比特提供范围从 0 到 15 中的数，其与 8 相加来提供范围从 8 到 23 中的旋转计数。$\mathcal{I}_i$旋转计数被应用于至$\mathcal{M}$的输入，并且$\mathcal{O}_i$旋转计数被应用于来自$\mathcal{M}$的输出。

[0583] 当$\mathcal{M}$的输入和输出在下一步骤中被置换时，这些旋转不被置换。

[0584] 1115 选择$p \approx s(\log_2 s)^2$选择子对$\mathcal{A}_1, \ldots, \mathcal{A}_p, \mathcal{B}_1, \ldots, \mathcal{B}_p, \mathcal{U}_1, \ldots, \mathcal{U}_p, \mathcal{V}_1, \ldots, \mathcal{V}_p$，每个类似于上面的 C，其仅提供足够$\mathcal{A}_i$到$\mathcal{B}_i$比较以及足够$\mathcal{U}_i$到$\mathcal{V}_i$比较，以通过§2.8的方法、通过控制其随机交换来分别控制至$\mathcal{M}$的输入和来自$\mathcal{M}$的输出的我们的随机置换。每个比较以约$\frac{1}{2}$的交换概率生成布尔决定（交换或不交换）。

[0585] 围绕针对$\mathcal{M}$的 s 输入和输出选择的功能性的逻辑定序是：初始旋转、然后初始置换、然后输入编码、然后矩阵映射，然后t输入输出功能性（$\mathcal{R}$部分功能性）、然后输出编码、然后输出置换、然后最后旋转。当选择子使用$\mathcal{M}$输入时，其以与$\mathcal{M}$所做相同的编码（即，M_{in}编码）来使用它们，所以不管任何置换，选择子始终以与$\mathcal{M}$所做精确相同的顺序精确地看到相同输入。

[0586] 注意，所有以上步骤在用于$\mathcal{M}$功能性的循环内；即，从初始旋转到最后旋转的所有操作在每个迭代上执行。

[0587] 作为结果，简化是可能的：例如，输入编码不需要单独针对$\mathcal{M}$和使用$\mathcal{M}$的输入的选择子完成；它们可以共享相同的经编码的值。

[0588] 1125 我们现在继续由 s 输入输出部分和 t 输入输出部分（$\mathcal{M}$功能性部分和$\mathcal{R}$功能性部分）构成的内部 FII 实现。

[0589] 使用§6.2.1的一般方法，选择具有对应的函数$\mathcal{C}'$的$1 \times s$选择子矩阵C'，其取与$\mathcal{M}$相同的输入并且具有输入编码L'_{in}和输出编码C'_{out}；即，$C' = C'_{out} \circ C' \circ M_{in}$。（对应的$I_K^{-1}$区段将具有形式为$C'_{out} \circ C' \circ L'^{-1} \circ L'^{-1}_{out}$的选择子——其当然将被简化。）

取$\mathcal{C}'$的输出的两个高阶比特并加 2 以形成范围从 2 到 5 中的迭代计数。比该迭代小一的迭代计数是范围从 1 到 4 中的数，其是在 s 输入输出、$\mathcal{M}$部分循环的一次迭代期间，$\mathcal{R}$功能性的输出（取 t 个输入并产生 t 个输出）具有直接反馈到其输入的其输入并总体被再次执行的次数。即，在针对中 s 输入输出部分的一次迭代中，针对 t 输入输出的所有迭代被执行，所以如果 s 部分迭代四次并且 t 部分迭代计数为三，则 t 部分被重复 12 次：针对每个 s 部分迭代三次。

[0590] 1130 选择 $2t$ 选择子函数 $\mathcal{I}'_1,\ldots,\mathcal{I}'_t$ 和 $\mathcal{O}'_1,\ldots,\mathcal{O}'_t$，每个类似于上面的 $\mathcal{C}'$。这些中的高阶四个比特提供范围从 0 到 15 中的数，其与 8 相加来提供范围从 8 到 23 中的旋转计数。$\mathcal{I}'_i$ 旋转计数被应用于至 $\mathcal{R}$ 的输入，并且 $\mathcal{O}'_i$ 旋转计数被应用于来自 $\mathcal{R}$ 的输出。

[0591] 当 $\mathcal{R}$ 的输入和输出在下一步骤中被置换时，这些旋转不被置换。

[0592] 1135 选择 $q \approx t(\log_2 t)^2$ 选择子对 $\mathcal{A}'_1,\ldots,\mathcal{A}'_q$、$\mathcal{B}'_1,\ldots,\mathcal{B}'_q$、$\mathcal{U}'_1,\ldots,\mathcal{U}'_q$ $\mathcal{V}'_1,\ldots,\mathcal{V}'_q$，每个类似于上面的 $\mathcal{C}'$，其仅提供足够 $\mathcal{A}'_i$ 到 $\mathcal{B}'_i$ 比较以及足够 $\mathcal{U}'_i$ 到 $\mathcal{V}'_i$ 比较，以通过 §2.8 的方法、通过控制其随机交换来分别控制至 $\mathcal{R}$ 的输入和来自 $\mathcal{R}$ 的输出的我们的随机置换。每个比较以约 $\frac{1}{2}$ 的交换概率生成布尔决定(交换或不交换)。

[0593] 围绕针对 $\mathcal{R}$ 的 t 输入和输出选择的功能性的逻辑定序是：初始旋转、然后初始置换、然后输入编码、然后矩阵映射，然后输出编码、然后输出置换、然后最后旋转。当选择子使用 $\mathcal{R}$ 输入时，其以与 $\mathcal{R}$ 所做相同的编码(即，R_m 编码)来使用它们，所以不管任何置换，选择子始终以与 $\mathcal{R}$ 所做精确相同的顺序精确地看到相同输入。

[0594] 注意，所有以上步骤在用于 t 输入输出($\mathcal{R}$ 部分)功能性的循环内；即，从初始旋转到最后旋转的所有操作在每个迭代上执行。

[0595] 作为结果，简化是可能的：例如，输入编码不需要单独针对 $\mathcal{R}$ 和使用 $\mathcal{R}$ 的输入的选择子完成；它们可以共享相同的经编码的值。

[0596] 6.3 混淆 f_K 或 f_K^{-1} 实现。以下方法被用于模糊程序实现 f_K 或 f_K^{-1}，其中实现具有以上 §6.2 中给出的结构。

[0597] 以下节中的变换一个接一个地执行，除了在这些节的主体中另外指出的情况以外。

[0598] 6.3.1 清除。在 §5.2.1、§5.2.2 和 §5.2.3 中列出的清除按照需要执行，如在标记 II 实现中那样。

[0599] 6.3.2 *独特动态值的哈希插入和生成。§5.2.4 的变换被执行，但是使用取所有输入的 1×12 矩阵。否则，这与对应的标记 II 步骤非常类似。

[0600] 6.3.3 宏指令扩展。这如在标记 II 实现中那样完成。

[0601] 6.3.4 来 - 自插入。这如在标记 II 实现中那样完成。注意，在标记 III 中，所有控制流存在以创建嵌套的每区段循环。

[0602] 6.3.5 随机指令重编码。我们注意到，在生成实现的区段中所采用的函数索引的交织(FII)中，我们已经将输入和输出分别划分成宽度为 r-s-t 的可能不规则的组。在 f_K 和 f_K^{-1} 中，

- r 输出仅取决于 r 输入；
- s 输出取决于 r 和 s 输入；以及
- t 输出取决于 r、s 和 t 输入。

[0603] 其中针对 r 和 s 之间的 FII 的选择子计算被考虑为 s 计算的部分，并且针对在 s 和 t 之间的 FII 的选择子计算被考虑为 t 计算的部分。注意，以该考虑，s 输出不取决于 r 输出，并且 t 输出不取决于 r 和 s 输出。

[0604] 如果实现不是 SSA 形式，则将其转换成 SSA 形式(参见 §2.10.1)，并且移除冗余的 MOVE 指令(参见 §5.2.1)。

[0605] 我们现在对每个区段自身进行拓扑排序，由此随机混合 r、s 和 t 指令序列。

[0606] 我们类似地对初始混合自身进行拓扑排序，并且对最后混合自身进行拓扑排序。

[0607] 我们串接排序后的定序：初始混合、区段 1、区段 2……区段 k、最后混合。在该串接的定序中创建表示“在前”关系的新关系 R。通过以下方式来创建新关系 R'：随机一致地移除在每两个弧 $(x,y) \in R$ 中的一个，并将 R' 与执行约束合并以形成总体“在前”关系，再次对整个序列进行拓扑排序。

[0608] 6.3.6 *数据流复制。这如在标记 II 实现中那样完成(参见 §5.2.8)。

[0609] 6.3.7 *随机交叉连接。这如在标记 II 实现中那样完成(参见 §5.2.9)。

[0610] 6.3.8 *检查插入。这如在标记 II 实现中那样完成(参见 §5.2.10)，具有以下改变：在用于检查的候选布置间，以 $\frac{1}{2}$ 概率选择在当前区段内的候选(在这样的候选存在的情况下)，并且以 $\frac{1}{2}$ 概率选择之后区段中的候选(在这样的候选存在的情况下)。作为该改变以及在 §6.3.5 中的修改后的重定序方案的结果，以高概率，r,s,t 区段的全部通过插入的检查交叉连接并且被使得取决于彼此。

[0611] 6.3.9 代码转换。这如在标记 II 实现中那样完成(参见 §5.2.11)。

[0612] 6.3.10 寄存器最大化。这如在标记 II 实现中那样完成(参见 §5.2.12)。

[0613] 6.3.11 *存储器混洗。这如在标记 II 实现中那样完成(参见 §5.2.13)。注意，因为我们具有循环但没有 *if then else*，所以 W_1 被最小地生成，这消除了可能在标记 II 实现中产生的一些异常。

[0614] 6.3.12 最后清除和代码发射。这些如在标记 II 实现中那样完成(参见 §5.2.15)。

[0615] 7. 掺合与锚定技术

如果互逆基函数的对的成员可以锚定到采用其的应用和该应用驻留于的平台，并且如果它的数据和代码可以与它形成其一部分的应用的数据和代码掺合，则它的值大大增加，意义在于不同种类的数据或代码之间的边界变模糊。

[0616] 这样的锚定和掺合的效果是：

(1)挫败代码和数据提升攻击，

(2)通过混淆边界的精确位置来挫败输入点、输出点以及其他边界攻击，以及

(3)增加保护性代码和数据与它们周围的上下文代码和数据之间的相关性，由此通过在篡改下增加的脆性而增加篡改抵抗。

[0617] 要解决的数据和代码边界的种类为：

(1)输入边界，其中未编码的数据必须被编码并被从其未受保护域带到受保护域(参见

§7.3),

(2)输出边界,其中受保护数据必须被解码并被从受保护域带到未受保护域(参见§7.3),

(3)进入边界,其中控制从未受保护代码传递到受保护和被混淆代码(参见§7.4),

(4)退出边界,其中控制从受保护和被混淆代码传递到未受保护代码(参见§7.4),

(5)分离边界,其中数据从来自多个变量的熵跨可设置大小的比特矢量均匀混合的形式改变为来自单独变量的熵更加隔离的形式,但是仍然被编码,并且计算在这些更加隔离的变量上执行,以及

(6)混合边界,其中数据从来自单独变量的熵被编码但相对隔离的形式(一般包含在这样的变量上的计算的结果)改变到来自多个变量的熵跨可设置大小的比特矢量被均匀混合的形式。

[0618] 保护分离和混合边界的挑战在于,在分离之后或混合之前在其上执行计算的数据频繁地来自于其他站点,在所述其他站点中,数据也被分离成相对少的变量。这允许其中隔离的变量被扰动的攻击,结果是:在混合和重新分离之后,隔离的变量响应于该扰动,从而向在做出响应的站点处的那些揭露在扰动站点处的值的连接。

[0619] 除了以上掺合形式之外,我们寻求借助于互锁技术将代码和数据锚定到它们的上下文,包括:

(1)数据相关系数,其中在代码中的某些代码站点处的数据流提供用于计算系数的变量,所述系数控制在代码中的后续执行的代码站点处的编码(参见下面的§7.1),以及

(2)具有交叉检查和交叉俘获的数据流复制,其中数据流的某些部分被复制(但是具有不同编码),数据流链路在副本间被交换,并且注入计算,如果副本对于它们的未编码的值相匹配,则这没有净效应,但是如果副本对于它们的未编码的值未能匹配,则这导致计算敏锐地失败或降级,

(3)数据流恶化和修复,其中错误在某些代码站点处被注入到数据流中,并且这些错误在后续执行的代码站点处被纠正,

(4)控制流恶化和修复,其中只有执行代码需要处于正确地可执行状态,只要作为其执行的部分其确保后续状态的正确可执行性即可——实际上,存在包括当前执行代码的移动的正确性窗口,并且当除了在进入之前被纠正以外离开时代码恶化——经由到诸如例程变量、情况指标等的数据的改变,以避免在自修改代码中固有的问题,即,所有这样的恶化应当影响在控制中使用的数据而不是实际代码自身。

[0620] (5)共享黑板,其中多个代码段利用动态数据识别编码的实例,其中动态解决的存储与持续的数据混洗和重编码在多个代码段间共享,使得攻击者更难从属于其他代码段的数据流中分离一个代码段的数据流,

(6)并行功能,其中用于执行某个功能的代码与执行一个或多个其他功能的代码交织,使得两个或更多功能被单个控制线程并行执行,这(与上面的其他技术相组合)使分离用于两个功能的代码是可能的,以及

(7)子集团和段,其中我们部署团和段控制流保护专利(US 6,779,114)的功能性的子集,包括可切换的非常大的例程,其将多个例程组合成单个例程。

[0621] 7.1 数据相关系数。检查用于置换多项式逆的方程(参见§2.3.2、§2.3.3和§2.3.4,我们注意到乘法环逆在字环(当前计算机上典型为2^{32}或2^{64})上)被广泛使用。(它们是方程中分式中的支配者:例如,a/c^4意味着意指$a(c^{-1})^4$的ac^{-4},其中c^{-1}是c的乘法环逆。)

对于具有w比特字的机器,c和c^{-1}是使得$c \cdot c^{-1} = 1$的两个数,其中$\cdot$是在环内的乘法运算操作,即,其是两个w比特二进制字的乘法,其中如C和C++中那样忽略上溢。

[0622] 尽管我们可以通过采用外延的欧几里德算法[15,26]来通过计算找到这样的逆,但是这是不期望的,因为该算法是可容易识别的。因此,我们需要另外的手段来将输入数据提供的熵中的一些转换成用于置换多项式的随机系数的选择。

[0623] 我们记住,沿着这些线的方法预先随机地选择数:

$$a_0, a_1, a_2, \cdots, a_{w-1}$$

其中每个a_i是在范围从3到$2^w - 1$(包含性的)中的奇数,所有a_i是成对独特的,并且在列表中不存在对a_p, a_q使得$a_p \cdot a_q = 1$。我们将还采用预先使用上面提到的外延的欧几里德算法找到的它们的乘法逆

$$a_0^{-1}, a_1^{-1}, a_2^{-1}, \cdots, a_{w-1}^{-1}$$ 。

[0624] 然后对于从在CB函数内的早期计算选择的任何随机非零字值v,我们选择使用v的比特的产物c:如果2^i在v中被设置,则a_i在产物中。这给出了从1到w因数的产物,其逆通过再次使用v来找到:如果比特2^i在v中被设置,则a_i^{-1}在给出逆c^{-1}的产物中。

[0625] 如果w=32,则这向我们给予非常大量的潜在系数和逆系数。实际上,该数量如此之大,以至于我们可以选择仅使用v的部分——即,通过某个较小的数替换w并具有较小数量的a_i和它们的逆——其可能仍然是足够的:取代32的18比特仍将允许在200,000个系数+逆系数对上进行选择。

[0626] 注意,我们仅提供了用于生成奇系数的手段。其他种类的系数更容易生成,因为或者我们仅需要它们的加法逆(偶元素没有乘法逆)。为了生成为偶数的系数,我们简单地使所生成的值v翻倍。为了创建其平方为0的系数,我们简单地将v在逻辑上左移位$\lfloor \frac{w}{2} \rfloor$个位置(即,我们将其乘以$2^{\lfloor \frac{w}{2} \rfloor}$,其中上溢被忽略)。

[0627] 7.2 针对编码强度的细微控制。在当前代码转换中,存在称为数据流级别和控制流级别的设置,其在正常情况下从0运转到100,从而指示应当对数据或控制流进行多强的编码。

[0628] 传统地,用于影响该表面地细粒度控制的机制具有两个变体:

(1)在数据或控制流级别中的某些特定数字阈值处出现的行为上的突然不同,使得在阈值以下不应用某个变换,并且在阈值以上应用该变换,以及

(2)在针对在某个代码片段上执行某个变换的概率阈值中的细粒度不同,使得在较低数据流级别,伪随机变量可能必须下降到0.8以上以导致其被变换,而在较高数据流级别其可能仅必须下降到0.2以上以导致变换。

[0629] 我们对于方法(1)没有问题,但是我们可以对(2)进行改进。对于方法(2)的问题

是：只是偶然，实现的实际级别可能未下降到意图级别附近。我们因此建议以下改进。

[0630] 我们保持对要覆盖的总站点的运行结算、目前为止覆盖的站点的运行结算，以及有多少接收的概率受控变换（包括的站点）以及有多少没有（被排除的站点）。当包括的站点的比率在期望比率以下时，我们将执行变换的概率增加到其正常值以上，并且当其在期望比率以上时，我们将执行变换的概率降低到在其正常值以下。针对增加和下降的程度的合适设置可以通过实验来计量。这可以有效地使得用于受影响代码区的实际比率被保护以紧密跟踪期望比率，而非由于几率效应而偏离该比率。

[0631] 当潜在站点的总数大时，这将具有其最佳效果。如果仅有几个站点存在，缺乏海量代码复制以增加站点的有效数量，则可以运用很少的细粒度控制。

[0632] 这容易外延到具有多于两个选择的情况。考虑例如在$\mathbb{Z}/(2^{32})$上（或最近在$\mathbb{Z}/(2^{64})$上，更强大的平台）的置换多项式编码的使用。如果我们在无编码或次数 1 到 6 的编码间改变，则存在七个可能的选择要覆盖，在它们当中，我们根据期望的比率来分派概率。相同的原理应用：如果我们正得到太少的内容，则我们将其概率上推；如果我们正得到太多内容，则我们将其概率下降。

[0633] 7.3 输入和输出边界。在输入边界处，未编码的数据必须被编码并被从其未受保护域带到受保护域。在输出边界处，受保护数据必须被解码并被从受保护域带到未受保护域。

[0634] 这是用于部署针对数据流的编码强度控制（参见§7.2）的合适位置。测量在数据流图中的多个图弧中的数据流距离，其中弧将值产生操作连接到消耗它的操作，我们如下继续。

[0635] （1）保护在其选择的强度（通常相当高）处的实现。

[0636] （2）针对输入边界和输出边界二者，保护远离在实现内部的操作$1, 2, 3, \ldots, k$个弧的操作，所述操作具有减弱的强度，直到达到周围代码的正常代码转换强度为止。

[0637] 这要求我们能够计量这样的距离，这要求来自代码转换器的一些额外支持以添加这样的选择性距离信息，并通过如在§7.2概述的那样控制编码强度来对其作出响应。

[0638] 应用于输入／输出边界的附加保护是数据相关系数，用于增加基函数进入处的计算与提供输入的代码的相关性，并用于增加接收基函数输出的计算与提供那些输出的实现内的代码和共享黑板的相关性（如果数据可以通过经由共享黑板而进入和离开——动态数据识别编码的实例——则攻击者为了该数据而跟随数据流要难得多）。

[0639] 7.4 进入和退出边界。典型地，在其处实现接收其输出的点紧跟在其处控制进入实现的点，并且在其处实现提供其输出的点紧在其处控制离开实现的点之前。

[0640] 作为结果，在§7.3中的保护也保护进入和退出边界。然而，该实现将典型地比常规代码转换的代码具有更强的控制流保护。

[0641] 因此，我们需要对进入执行细粒度的逐步递增和对控制流级别的退出执行逐步减弱。这里我们对距离的度量是要沿着导致进入点（对于进入而言）或离开退出点（对于退出而言）的最短路径执行的 FABRIC 或机器代码指令的估计数量，其中对于针对进入和退出的每一个的比方说一百或两百个指令单元的距离进行掺合。

[0642] 这将是绝佳的位置来部署控制流恶化和修复，以将靠近进入的代码和保护性进入代码以及保护性退出代码和从退出移动离开的代码系在一起，以增加在保护性进入和退出的附近中的保护级别。

[0643] 7.5 分离和混合边界。其中我们遭遇分离和混合边界的一般情形是这样的情形，其中结构化的数据以轻微编码或未编码的形式从实现输出，或者轻微编码或未编码的结构化数据进入实现，或者本发明的实现的一部分用于包夹我们希望隐藏的做决定计算。

[0644] 分离和 / 或混合的效果是我们在分离之后或在混合之前具有潜在地被曝露的数据，从而产生攻击点。隐藏，除了在 §7.3 中已经覆盖的针对这些情形的相关保护之外，我们需要针对我们需要在用作分离或混合函数的基函数之间包夹的任何计算的更强的保护。

[0645] 如果决定基于检查口令或者一些类似的相等比较，则我们强烈推荐 §A 的方法作为用于其保护的最佳选择。然而，我们很少如此幸运。

[0646] 更常见的情况是我们需要执行一些算法、一些比较、一些按比特布尔运算等等。为了这些，我们推荐以下保护（参见 §7）：

（1）首先且最重要的，具有交叉检查和交叉俘获的数据流复制用于大量增加在初始基函数与决定代码以及在决定代码与最后基函数之间的数据相关性；

（2）数据流相关系数的文字使用：在决定块中的系数由在前基函数设置，并且在随后基函数中的系数由决定块中的代码设置；

（3）共享黑板（动态数据识别编码阵列）作为站点的使用用于从初始基函数到决定代码以及从决定代码到最后基函数的通信；以及

（4）如果可能的话，并行功能，使得决定块与其他不相关代码进行混合，使得攻击者难以分析和从代码进行区分，以所述代码，其通过交织它们的计算而并行计算。

[0647] 7.6 一般保护。某些保护可以在每个边界处和在边界之间应用以便进一步在其中基函数被部署的上下文中保护代码，即控制和数据流恶化和修复、并行功能以及子集团和段。在可行的情况下，这些添加的保护将增加攻击者面对的分析难度，具体地，它们将呈现静态分析不可行以及动态分析是高代价的。

[0648] 7.7 示例性保护场景。数据以经由基函数编码的形式被提供，使得信息跨其整个状态矢量而被涂覆。因此编码的数据包括

（1）128 比特密钥

（2）具有各种字段的 128 比特数据结构，一些字段仅为几比特宽，一个字段为 32 比特宽，并且一些字段高达 16 比特宽。

[0649] 计算要对数据结构的字段被执行并且对在当前平台上的信息被执行，作为其的结果，或者密钥将以已供使用的形式被递送（指示当前平台信息加上递送的数据结构导致释放所述密钥的决定）或者与密钥相同大小的无意义串将以表现为已供使用但是实际上将失败的形式被递送（指示当前平台信息加上递送的数据结构导致不释放所述密钥的决定）。

[0650] 攻击者的目标是获得 128 比特密钥，而不管字段的内容和在当前平台上的信息。防御者的目标是确保在“是”决定时密钥被正确递送，除非防御者的实现被篡改，但是在其中决定将是在没有篡改的情况下的“否”的情形中，密钥是攻击者不可获得的。

[0651] 这捕捉到保护系统的主要掺合需要：存在输入、输出、进入、退出、分离以及混合边

界要保护。

[0652] 7.8 用掺合实现保护。这里，我们针对如§7.2到§7.6中所提出的在p.67上在§7.7中描述的保护场景，通过§7开始列出的子集团和段描述从数据流相关系数实现保护。

[0653] 7.8.1 开始配置。我们的开始配置包括保护系统核心的代码转换器中间表示、包含应用输入256比特值(包含编码形式的128比特经编码的密钥)到其的128比特X256比特函数，以及128比特数据结构并且应用从其接收已供使用的128比特不同编码的密钥。

[0654] 该核心包括：

(1)进入256比特×256比特基函数，其接受其中熵跨整个256比特混合的256比特的，将此解码为具有128比特经编码的密钥(通过某个其他128比特×128比特基函数预先编码)的结构和具有平滑(未编码)形式的扩展字段的128比特数据结构；

(2)决定块，其接受128比特数据结构和128比特密钥，在128比特数据结构的字段上执行计算，决定是否释放密钥，以及向第二基函数提供128比特经编码的密钥自身(如果决定是“继续”)或使用密钥的值以及形成作为熵源的128比特数据结构的进一步信息，并向第二128比特×128比特基函数提供经编码的密钥或相同宽度的无意义的值；

(3)退出128比特×128比特基函数，并返回已在一些白盒密码函数(例如，AES-128)中使用的不同编码的密钥。

[0655] 进入和退出基函数根据标记III决定(参见§6)来构造。该核心在其包含例程中是内联代码；即，其不通过例程调用进入也不通过例程返回退出：相反，周围功能性被包括在包含该周围功能性和该核心的例程内。

[0656] 核心和应用的组合被称为程序。

[0657] 7.8.2 *细分区段输入和输出矢量。以下材料是示例性的：更广泛选择存在。这里，我们选择双重递归函数索引的交织，从而产生对区段的三部分划分。也可以是单重递归(两部分划分)、三重递归(四部分划分)或阶n递归((n+1)部分划分)。

[0658] 在§6.2.3中，划分针对在12字宽(384比特I/O)标记III实现中的I/O矢量给出。根据以上，我们具有8字宽进入基函数和4字宽退出基函数。我们将进入区段细分如下：

2-2-4 2-3-3 3-2-4 3-3-2 4-2-2 。

[0659] 如果我们以从0到3的顺序对上面四个配置编号，则我们生成的针对每个区段所选的配置的号码静态地通过rand(4)来选择；即，我们在构造时随机一致地从以上四个可能性中进行选择。

[0660] 对于退出基函数，区段被细分如下：

1-1-2 1-2-1 2-1-1 。

[0661] 如果我们以从0到2的顺序对上面三个配置编号，则我们生成的针对每个区段所选的配置的号码静态地通过rand(3)来选择；即，我们在构造时随机一致地从以上三个可能性中进行选择。

[0662] 7.8.3 *距离度量。我们利用从操作到核心输入以及从核心输出到操作的距离的测量。存在四个度量：两个用于数据流距离并且两个用于控制流距离。

[0663] ----------

*数据流距离。我们收集下至 -200 的输入距离和上至 +200 的输出距离。超过该点，我们可以忽略更大的距离，并且应用启发式方法来避免计算它们。

[0664] 核心中的每个计算指令与核心的距离是零。(计算指令是输入一个或多个值或者输出一个或多个值的指令。)

每个其他计算指令(CI)与核心的距离是负(如果其提供影响被核心消耗的值的值)或正(如果其消耗被核心产生的值影响的值)。

[0665] 我们假设，对于大多数部分，二者都不为真；即，核心不在其中核心被重复采用的循环的体中，或者其在循环中，但是该循环充分外延以至于我们可以忽略馈送至核心中的被来自之前核心执行的输出影响的任何信息。然而，指令可以驻留在可以既在核心执行之前又在之后调用的例程中，在该情况中，数据流距离是包括其输入距离和其输出距离的对。

[0666] 输入距离被确定如下。

[0667] • 如果 CI 输出是到核心的直接输入的值或者加载是到核心的直接输入的值(核心接受所述直接输入作为数据流边沿(即，作为“虚拟寄存器”))或者存储核心从存储器加载的输入，则其输入距离为 -1。否则，

• 如果 CI x 输出是到具有 -k 的输入距离(并且归因于如上所提到的在例程中的指令既在核心之前也在之后被调用，也可能是+k 的输出距离)的 y CI 的直接输入的值，或者 x 加载为通过这样的 y 输入的值或者存储这样的 y 从存储器加载的输入，则其输入距离为 -k-1。

[0668] 当以上考虑给予指令多个独特输入距离时，最接近零的那个是正确的。

[0669] 输出距离被确定如下。

[0670] • 如果核心输出是到 CI 的直接输入的值或者加载是到 CI 的直接输入的值(CI 接受所述直接输入作为数据流边沿(即，作为“虚拟寄存器”))或者存储 CI 从存储器加载的输入，则其输入距离为 +1。否则，

• 如果具有输出距离 k (并且归因于如上所提到的在例程中的指令既在核心之前也在之后被调用，也可能是-k 的输入距离)的 CI x 输出是到 y CI 的直接输入的值，或者这样的 CI x 加载为通过这样的 y 输入的值或者存储这样的 y 从存储器加载的输入，则其输出距离为 +k+1。

[0671] 当以上考虑给予指令多个独特输出距离时，最接近零的那个是正确的。

[0672] 该度量完全忽略控制流。以值返回指令被认为是用于该目的的加载。向例程内的变量输入值的例程进入指令被认为是用于该目的的存储指令。

[0673] ----------

*控制流距离。我们收集下至 -200 的进入距离和上至 +200 的退出距离。超过该点，我们可以忽略更大的距离，并且应用启发式方法来避免计算它们。

[0674] 我们将指令视为通过程序的控制流图中的有向弧来连接，具有两个外发弧(如果测试条件为真或假，则引导到后继)的条件分支，以及具有通过控制索引选择的多个后继的索引分支(case 或 switch 语句分支)，所述控制索引针对控制结构的 case 标签而被测试。。对于例程返回指令，通过从其调用例程的站点来确定其后继；即，它们是可以紧跟在从例程返回后执行的所有指令，并且返回指令被认为是至那些返回后指令的索引分支。

[0675] 在核心内的任何指令具有与核心相距为零的控制流距离。如上那样，我们假设无循环场景，其中涉及核心的任何循环是足够大尺度的并且足够少发生的以允许我们忽略它。然而，在控制流距离的情况中，指令可以驻留在可以既在核心执行之前又在之后调用的例程中，在该情况中，控制流距离是包括其进入距离和其退出距离的对。

[0676] 进入距离被确定如下。

[0677] • 如果指令具有在核心中的后继指令或者是具有在核心中的目的地的分支，则其进入控制流距离为 -1。否则，

• 如果指令 x 具有紧后继指令 y，y 具有 -k 的进入控制流距离（并且归因于如上所提到的在例程中的指令既在核心之前也在之后被调用，也可能是 $+k'$ 的退出控制流距离），或者 x 是其目的指令是这样的 y 的一个分支，则其进入控制流距离为 -k-1。

[0678] 当以上考虑给予指令多个独特进入距离时，最接近零的那个是正确的。

[0679] 退出距离被确定如下。

[0680] • 如果核心指令具有在核心外部的后继指令或者是具有在核心外部的目的指令的分支，则在核心外部的该指令具有 +1 的退出控制流距离。否则，

• 如果具有 +k 的退出控制流距离（并且归因于如上所提到的在例程中的指令既在核心之前也在之后被调用，也可能是 $-k'$ 的进入控制流距离）的指令 x 具有紧后继指令 y，或者如果这样的 x 分支到指令 y，则 y 具有为 +k+1 的退出控制流距离。

[0681] 当以上考虑给予指令多个独特退出距离时，最接近零的那个是正确的。

[0682] 7.8.4 清除。在 §5.2.1、§5.2.2 和 §5.2.3 中列出的清除按照需要执行，如在标记 II 实现中那样，不仅用于进入和退出基函数实现，而且也用于决定块和应用。

[0683] 7.8.5 * 独特动态值的哈希插入和生成。取代每基函数执行 §5.2.4 的变换一次（即，针对进入函数执行它，以及单独地针对退出函数执行它）的是，有时在计算进入基函数之前，我们使用 1×16 矩阵在应用内执行该变换，所述 1×16 矩阵的输入是从应用中可用的数据选择的。我们使用它来创建用于动态数据识别编码的阵列，其将服务于该应用，进入和退出基函数以及决定块二者，使得它们都使用一个共享的黑板。

[0684] 7.8.6 宏指令扩展。这如在标记 II 和标记 III 实现中那样完成。

[0685] 7.8.7 来 - 自插入。这如在标记 III 实现中那样完成，但是外延到与核心相距进入距离或退出距离的具有 100 或更小绝对值的每个分支；即其良好地外延超过核心中的透明盒实现的限制。

[0686] 7.8.8 * 控制流恶化和修复。作为代码处理的部分，代表被变平：在像 switch 语句的构造中的目的地点做出分支标签，并且通过分支到该 switch、向其传递对应于引导到期望目的地的 switch case 标签的索引来到达目的地。

[0687] 这应当针对在核心中或在基本块中的所有代码完成，所述基本块具有与核心相距进入距离或退出距离的任何指令，所述距离具有 100 或更小的绝对值。

[0688] 我们考虑要在变量中 $v_1, \ldots, v_n$ 中存储的目的地的对应索引，所述变量对应于在控制流图中表示基本块 $B_1, \ldots, B_n$（经由标签进入并且经由最后分支、返回或调用退出）的节点。

[0689] 在变平之前，我们在以下约束下用总双射函数$L_i: \{1,\ldots,n\} \leftrightarrow \{1,\ldots,n\}$（$i = 1,\ldots,n$）来随机地标记在恶化区中的每个基本块。

[0690] （1）如果基本块B_i可以退出到$B_{j_1},\ldots,B_{j_m}$，则其标记L_i函数具有$L_i(j_k) = j_k$，$k = 1,\ldots,m$的属性。

[0691] （2）如果两个独特基本块B_i, B_j可以都退出到块B_k，则$L_i = L_j$。

[0692] （3）如果基本块B_i可以退出到块B_j，则在其处$L_i(k) \neq L_j(k)$的点k的数量通过以下来上限界：由可以退出到B_j的任何基本块处理的目的基本块的数量的四倍。

[0693] 在变平之后，每个基本块B_m在使得对于其前驱的L_i有$L_i(j) = v_j$（$j = 1,\ldots,n$）的状态中与变量$v_1,\ldots,v_n$一起进入。（这不意味着变量的状态是正确的，而仅意味着它们同意前驱的L_i）。然后继续交换变量使得对于每个变量，$L_m(j) = v_j$——这几乎无疑是与其一起进入的变量不同的$v_1,\ldots,v_n$的状态，但是鉴于约束，改变的数量具有合理的限界。

[0694] 因此，在达到基本块的最后时之前，变量以当前目的地是正确的但大多数其他通常是不准确的方式对应于目的地。

[0695] 7.8.9 随机指令重定序。我们注意到，如在实现的区段生成中所采用的函数索引的交织（FII）中，我们已经分别将输入和输出划分成宽度为r-s-t的可能不规则的组。在f_k和f_k^{-1}中，对于进入和退出基函数的每一个：

- r 输出仅取决于 r 输入；
- s 输出取决于 r 和 s 输入；
- t 输出取决于 r、s 和 t 输入；

其中针对 r 和 s 之间的 FII 的选择子计算被考虑为 s 计算的部分，并且针对在 s 和 t 之间的 FII 的选择子计算被考虑为 t 计算的部分。注意，以该考虑，s 输出不取决于 r 输出，并且 t 输出不取决于 r 和 s 输出。

[0696] 如果程序不是 SSA 形式，则将其转换成 SSA 形式（参见 §2.10.1），并且移除冗余的 MOVE 指令（参见 §5.2.1）。

[0697] 我们现在对在进入和退出基函数的每一个中的每个区段自身进行拓扑排序，由此随机混合 r、s 和 t 指令序列。我们类似地对初始混合自身进行拓扑排序，并且对每个基函数中的最后混合自身进行拓扑排序。我们同样对应用和决定块中的每个基本块（没有任何分支或例程调用或返回的代码的直线展开）进行拓扑排序。

[0698] 对于进入和退出基函数的每一个，我们串接排序后的定序：初始混合、区段 1、区段 2……区段 k、最后混合。在该串接的定序中创建表示“在前”关系的新关系 R。通过以下方式来创建新关系R'：随机一致地移除在每两个弧$(x,y) \in R$中的一个，并将R'与执行约束合并以形成总体“在前”关系，再次对整个序列进行拓扑排序。

[0699] 针对程序的指令重定序现在完成。

[0700] 7.8.10 * 具有交叉检查 / 俘获的数据流复制。用于这些变换的方法如在标记 II

中那样(参见§5.2.8、§5.2.9和§5.2.10),在标记 III 中具有修改(参见§6.3.8),但是其也针对附加代码段完成。

[0701] 具体地,除了其在进入和退出基函数内的正常使用之外,我们还针对在决定块内的数据流执行这些变换,包括信息从进入基函数的输出到决定块的输入的转移,以及信息从决定块的输出到退出基函数的输入的转移。

[0702] 存在对这些步骤的进一步改变以用于我们的掺合场景(参见§7.7),在下一节中覆盖。

[0703] 7.8.11 *决定隐藏。在决定块中,128 比特结构的字段被检查,对它们执行计算,并且通过-失败决定被达到并且作为值被递送。我们复制这些计算中的一些,使得由一对任意常数 c_{pass} 和 c_{fail} 之一构成的决定值被生成至少八次。代码转换将使这些值看起来是独特的。

[0704] 因为它们是副本,所以交叉链接和交叉检查应用于它们。具体地,我们可以假设它们将产生 c_{pass},并且在此基础上,当密钥中的数据流字输入到退出基函数时对该数据流字执行操作,所述退出基函数在通过时取消但是在失败时不取消。取消值可以利用来自结构的进一步的值(如果 $c_1 - c_2$ 取消,则 $(c_1 - c_2)c_3$ 也取消)。

[0705] 这与如在§7.8.10中的交叉链接和交叉检查的组合将使得密钥以数据相关的方式混乱地改变,但是将以高概率使得只要在结构上的决定块测试引导到“失败”决定,与密钥相同大小的无意义的值就被递送到跟随退出基函数的应用代码。(该方法与§A中的口令检查技术相关)。

[0706] 7.8.12 代码转换。这如在标记 II 实现中那样完成(参见§5.2.11),但是具有以下改变。

[0707] 我们如下划分保护级别:

(1)使用线性映射的有限环保护。

[0708] (2)使用二次多项式的置换多项式保护;

(3)使用二次多项式元素的 2 矢量函数矩阵保护。

[0709] 在核心中的代码以级别 3(强)保护。在具有不超过 100 的绝对值的输入距离或输出距离外的代码以级别 2(中等)保护,以及应用的剩余部分以级别 1(弱)保护。

[0710] 另外,代码转换如下利用数据相关系数。

[0711] (1)在引导直到进入基函数的应用代码中得到的常数设置在进入基函数的代码转换中的系数的八分之一。

[0712] (2)在进入基函数中得到的常数设置在决定块的代码转换中的系数的四分之一。

[0713] (3)在决定块中得到的常数设置在退出基函数中的系数的四分之一。

[0714] (4)在退出基函数中得到的常数设置在从退出基函数接收输出的应用代码中的系数的至少八分之一。

[0715] 7.8.13 寄存器最小化。这如在标记 II 实现中那样执行(参见§5.2.12),但是针对整个程序。

[0716] 7.8.14 *动态数据识别编码(存储器混洗)。这如在标记 II 实现中那样执行(参见

§5.2.13),但是影响超过核心的代码。具体地,混洗后的存储器提供的共享黑板用于将来自应用的输入提供给进入基函数,并将来自进入基函数的输出提供给决定块,并且将来自决定块的输入提供给退出基函数,以及将来自退出基函数的输出提供给应用。

[0717] 7.8.15 最后清除和代码发射。这些如在标记 II 实现中那样执行(参见§5.2.15),但是针对整个程序。

[0718] 节 A *通过与混乱特征的相等性进行认证

假设我们具有其中认证如口令那样的应用:在 G(提供的值)匹配参考值Γ时,即当$G=\Gamma$时,认证成功。

[0719] 进一步假设我们关心当$G=\Gamma$时会发生什么,但是当不匹配时,我们仅认为无论如何,认证授权不再可行。即,当$G=\Gamma$时我们成功,但是如果$G\neq\Gamma$,则进一步计算可以简单地失败。

[0720] 认证相等性不受对两侧应用任何无损函数的影响:对于任何双射ϕ,我们可以等同地测试是否$\phi(G)=\phi(\Gamma)$。即使ϕ是有损的,如果仔细选择ϕ使得当$G\neq\Gamma$时$\phi(G)=\phi(\Gamma)$的可能性足够低,对相等性的认证也可以以高概率保持有效(例如,如在 Unix 口令认证中那样)。

[0721] 基于本文之前描述的技术,我们可以简单地执行这样的测试。我们之前描述了一种通过以下方式来挫败篡改的方法:复制数据流(参见§5.2.8)、随机地交叉连接复制实例之间的数据流(参见§5.2.9),以及执行经编码的检查以确保相等性未被损害(参见§5.2.10)。

[0722] 我们可以改编该方法来测试是否有$G=\Gamma$——在编码形式中,是否有$\phi(G)=\phi(\Gamma)$。我们注意到,产生$\phi(G)$的数据流已经沿着其中$G=\Gamma$的成功路径复制产生$\phi(\Gamma)$的数据流。我们因此对于该比较省略数据流复制步骤。然后,我们仅如在§5.2.9中那样交叉连接以及如§5.2.10中那样插入检查。通过将这些计算用作用于将来经编码的计算的系数,我们确保:如果$\phi(G)=\phi(\Gamma)$,则所有都将正常继续,但是如果$\phi(G)\neq\phi(\Gamma)$,尽管进一步计算将继续,但是结果将是混乱的并且其功能性将失败。此外因为ϕ是函数,所以如果$\phi(G)\neq\phi(\Gamma)$,我们可以肯定$G\neq\Gamma$。

[0723] 参考文献

[1] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, **Compilers: Principles, Techiques, and Tools**, 1986, Addison-Wesley, ISBN 0-201-10088-6.

[2] K.E. Batcher, *Sorting Networks and their Applications*, Proc. AFIPS Spring Joint Comput. Conf., vol. 32, pp. 307–314, 1968.

[3] See `en.wikipedia.org/wiki/Batcher_odd-even_mergesort` and `www.iti.fh-flensburg.de/lang/algorithmen/sortieren/networks/oemen.htm`

[4] O. Billet, H. Gilbert, C. Ech-Chatbi, *Cryptanalysis of a White Box AES Implementation*, Proceedings of SAC 2004 – Conference on Selected Areas in Cryptography, August, 2004, revised papers, Springer (LNCS 3357).

[5] Stanley T. Chow, Harold J. Johnson, and Yuan Gu, *Tamper Resistant Software Encoding*, US patent 6,594,761.

[6] Stanley T. Chow, Harold J. Johnson, and Yuan Gu, *Tamper Resistant Software – Control Flow Encoding*, US patent 6,779,114.

[7] Stanley T. Chow, Harold J. Johnson, and Yuan Gu, *Tamper Resistant Software Encoding*, US divisional patent 6,842,862.

[8] Stanley T. Chow, Harold J. Johnson, Alexander Shokurov, *Tamper Resistant Software Encoding and Analysis*, 2004, US patent application 10/478,678, publication US 2004/0236955 A1.

[9] Stanley Chow, Yuan X. Gu, Harold Johnson, and Vladimir A. Zakharov, *An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs*, Proceedings of ISC 2001 – Information Security, 4th International Conference (LNCS 2200), Springer, October, 2001, pp. 144–155.

[10] S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *White-Box Cryptography and an AES Implementation* Proceedings of SAC 2002 – Conference on Selected Areas in Cryptography, March, 2002 (LNCS 2595), Springer, 2003.

[11] S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *A White-Box DES Implementation for DRM Applications*, Proceedings of DRM 2002 – 2nd ACM Workshop on Digital Rights Management, Nov. 18, 2002 (LNCS 2696), Springer, 2003.

[12] Christian Sven Collberg, Clark David Thomborson, and Douglas Wai Kok Low, *Obfuscation Techniques for Enhancing Software Security*, US patent 6,668,325.

[13] Keith Cooper, Timothy J. Harvey, and Ken Kennedy, *A Simple, Fast Dominance Algorithm*, Software Practice and Experience, 2001, no. 4, pp. 1–10.

[14] Ron Cytron, Jean Ferrante, Barry K. Rosen, and Mark N. Wegman, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems 13(4), October 1991, pp. 451–490.

[15] *Extended Euclidean Algorithm*, Algorithm 2.107 on p. 67 in A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 2001 (5th printing with corrections). Down-loadable from `http://www.cacr.math.uwaterloo.ca/hac/`

[16] National Institute of Standards and Technology (NIST), *Advanced Encryption Standard* (AES), FIPS Publication 197, 26 Nov. 2001.
`http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[17] Alexander Klimov, *Applications of T-Functions in Cryptography*, PhD thesis under Adi Shamir, Weizmann Institute of Science, October, 2004, Theorem 5.3 p. 41.

[18] *DES*, §7.4, pp. 250–259, in A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 2001 (5th printing with corrections). Down-loadable from `http://www.cacr.math.uwaterloo.ca/hac/`

[19] G. Mullen and H. Stevens, *Polynomial functions (mod m)*, Acta Mathematica Hungarica 44(3–4), 1984, pp. 237–241.

[20] Harold J. Johnson, Stanley T. Chow, Yuan X. Gu, *Tamper Resistant Software – Mass Data Encoding*, US patent 7,350,085.

[21] Harold J. Johnson, Stanley T. Chow, Philip A. Eisen, *System and Method for Protecting Computer Software Against a White Box Attack*, US patent application 10/433,966, publication US 2004/0139340 A1.

[22] Harold J. Johnson, Philip A. Eisen, *System and Method for Protecting Computer Software Against a White Box Attack*, US patent application 11/020,313 (continuation in part of US patent application 10/433,966 – not yet found in USPTO publications database).

[23] Harold Joseph Johnson, Yuan Xiang Gu, Becky Laiping Chang, and Stanley Taihai Chow. *Encoding Technique for Software and Hardware*, US patent 6,088,452.

[24] Arun Narayanan Kandanchatha, Yongxin Zhou. *System and Method for Obscuring Bit-Wise and Two's Complement Integer Computations in Software*, US patent application 11/039,817, publication US 2005/0166191 A1.

[25] D. E. Knuth, *The art of computer programming, volume 2: semi-numerical algorithms*, 3rd edition, ISBN 0-201-89684-2, Addison-Wesley, Reading, Massachusetts, 1997.

[26] *Extended Euclid's Algorithm*, Algorithm X on p. 342 in D. E. Knuth, *The art of computer programming, volume 2: semi-numerical algorithms*, 3rd edition, ISBN 0-201-89684-2, Addison-Wesley, Reading, Massachusetts, 1997.

[27] Ronald L. Rivest, *Permutation Polynomials Modulo 2^w*, Laboratory for Computer Science, MIT, October 25, 1999.

[28] T. Sander, C.F. Tschudin, *Towards Mobile Cryptography*, pp. 215-224, Proceedings of the 1998 IEEE Symposium on Security and Privacy.

[29] T. Sander, C.F. Tschudin, *Protecting Mobile Agents Against Malicious Hosts*, pp. 44-60, Vigna, Mobile Agent Security (LNCS 1419), Springer, 1998.

[30] David R. Wallace, *System and Method for Cloaking Software*, US patent 6,192,475.

[31] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, *Handbook of Applied Cryptography*, §1.3, pp. 6-10. CRC Press, 2001 (5th printing with corrections). Down-loadable from http://www.cacr.math.uwaterloo.ca/hac/

节 C:在$\mathbf{Z}/(2^n)$上的多项式逆计算,程序部件变换以及透明盒密码*

在软件混淆和保护的理论与实践中,在$\mathbf{Z}/(2^n)$上并且更一般地在B^n上的变换扮演重要角色。透明盒研究不是例外。在此注解中,我们给出了计算给定置换多项式$f(x)$的置换逆$f^{-1}(x)$以及在$\mathbf{Z}/(2^n)$上的给定可逆多项式$f(x)$的乘法逆$f(x)^{-1}$的算法。对有效实现的特殊多项式函数与一般混淆原理协作的结果进行了讨论和呈现。

[0724] 我们还研究了在B^n上用多项式生成矩阵作为它们的行列式的算法,并且描述了使用在B^n上的置换多项式和矩阵函数来变换算术运算和数据阵列的算法。这些变换可以与现有的MBA变换合成来用于在一般透明盒密码设定中的软件操作的保护。给出示例来说明新算法。

[0725] 1 介绍和记号

令N是自然数的集合,并且Z是整数环。令$\mathbf{B}=\{0,1\}$。

[0726] 微处理器的算术逻辑单元的数学基础在以下代数系统中进行抽象。

[0727] 定义1。对于$n\in\mathbf{N}$,我们定义代数系统$(\mathbf{B}^n,\wedge,\vee,\oplus,\neg,\leq,\geq,>,<,\leq^s,\geq^s,>^s,<^s,\neq,=,\gg^s,\gg,\ll,+,-,*)$,布尔算术代数(BA代数)或BA[n],其中$\ll,\gg$表示左和右移位,*表示乘法,以及带符号比较与算术右移位由^{s}指示。n是代数的维度。

[0728] BA[n]包括布尔代数$(\mathbf{B}^n,\wedge,\vee,\neg)$、整数模数环$\mathbf{Z}/(2^n)$以及伽罗瓦域$\mathrm{GF}(2^n)$。

[0729] 注意,保护设计的非常基本的要求是使实现容易与应用代码混合。因此,在BA[n]上建立变换称为有效方法。我们还论证其是足够的,因为存在与BA[n]直接相关的足够数量的计算困难问题。

[0730] ----------

*版本:2012 年 1 月 30 日。

[0731] C.2 在 $\mathbf{Z}/(2^n)$ 上的多项式

令 $f(x)$ 是 $\mathbf{Z}/(2^n)$ 上的函数,其中 $n \in \mathbf{N}$。如果 $f(x)$ 可表示为 $\sum_{i=0}^{m} a_i x^i$,其中 $a_i \in \mathbf{Z}/(2^n)$, $i = 0, \ldots, m-1$ 并且 $m \in \mathbf{N}$,则 $f(x)$ 是多项式函数,或者多项式 $\mathbf{Z}/(2^n)[x]$ 是在环 $\mathbf{Z}/(2^n)$ 上的所有多项式的集合。

[0732] 令 $\mathbf{PP}(2^n)[x]$ 是在 $\mathbf{Z}/(2^n)[x]$ 中的所有置换多项式的集合。

[0733] 令 $x^{(k)}$ 是递降阶乘幂 $x(x-1)(x-2)\cdots(x-k+1)$,其中 $k \in \mathbf{N}$。任何多项式 $f(x)$ 可以表示为 $\sum_{k=0}^{m} a_k x^{(k)}$,其中 $x^{(0)}$ 是 1。

[0734] 对于 $i \in \mathbf{N}$,令 $v(i) = \sum_{r=1}^{\infty} \lfloor i/2^r \rfloor$。在 $\mathbf{Z}/(2^n)$ 上的每个多项式可以用形式 $f(x) = \sum_{j=0}^{w} a_j \cdot x^{(j)}$ 唯一表达,其中 $a_j \in \mathbf{Z}/(2^{n-v(j)})$,并且 w 是针对其 $2^n \nmid (w+1)!$ 但是 $2^n \mid w!$ 的唯一整数。由于该唯一性,w 被称为 $f(x)$ 的次数,由 $deg(f(x))$ 或 $\partial(f(x))$ 表示。

[0735] 注意,$v(i)$ 等于 $i!$ 的 2-adic 阶,其是 $i-s$,其中 s 是二进制数表示的 i 的所有数位之和,或者是 i 的汉明重量。这在本注解中的几个算法中非常有用。

[0736] 对于在 $\mathbf{Z}/(2^n)$ 上的多项式,它们的次数的上界是使得 $v(w+1) = n$ 但是 $v(w) < n$ 的数 w。假设 n 是 2 的幂并且 $n = 2^t$。因为 $v(n+1) = v(2^t+1) = 2^t + 1 - 2 = 2^t - 1 < n$, 并且 $v(n+2) = v(2^t+2) = 2^t + 2 - 2 = 2^t = n$,所以我们具有 $w = 2^t + 1 = n + 1$。例如,在 $\mathbf{Z}/(2^{32})$ 上的多项式,最高可能次数是 33。

[0737] 由于在 $\mathbf{Z}/(2^n)[x]$ 中的多项式的最高次数是约 $\ln(n)$ 的事实,与有限域 $GF(2^n)$ 上的多项式相比,这大大地减小了计算代价。

[0738] 在环上存在大量置换多项式。$\mathbf{Z}/(2^n)[x]$ 的基数是 $2^{n(n+2)-\sum_{k=1}^{n+1} v(k)}$。八分之一或 $2^{n(n+2)-\sum_{k=1}^{n+1} v(k)-3}$ 是置换。对于 $n = 32, 64$,分别存在 $2^{2037}, 2^{2227}$ 个置换。

[0739] C.3 置换多项式

对于给定多项式,$f(x) = \sum_{k=0}^{n+1} a_k x^k \in \mathbf{Z}/(2^n)[x]$,如果且仅如果 a_1 是奇数且 $\sum_{s=1} a_{2s+1}$ 和 $\sum_{s=1} a_{2s}$ 二者都是偶数,其才是置换。令人感兴趣的观察是在递降阶乘表示 $f(x) = \sum_{k=0}^{n+1} b_k x^{(k)}$ 中,条件变成 b_1 是奇数且 b_2 和 b_3 二者都是偶数。

[0740] 在本节中,我们提供计算置换逆 $f^{-1}(x)$ 的有效算法,其也称为合成逆。

[0741] C.3.1 计算置换多项式的原像(根)

对于给定置换多项式,我们具有计算其原像的算法。

[0742] 命题 1。令 $y = f(x) = \sum_{i=0}^{n+1} a_i x^i$ 是在 $\mathbf{Z}/(2^n)$ 上的置换多项式。对于任何给定

值$\beta \in Z/(2^n)$，我们可以提供以下步骤找到$\alpha \in Z/(2^n)$使得$f(\alpha)=\beta$：

1. 输入$f(x)$和β；

2. $\alpha=0$；

3. α的第 0 比特是$\beta-a_0$的第 0 比特；

(a) 对于从 1 到 n-1 的 i

(b) α的第 i 比特是$\beta-a_0-(a_1-1)\alpha-\sum_{i=2}^{n+1}a_i\alpha^i$的第 i 比特；

4. 输出α。

[0743] 该计算是正确的，因为对于置换多项式$f(x)$，$(f(x)-x)$的第 i 比特完全由x在$i=0,1,\cdots,i-1$处的比特值和$f(x)$的系数确定。

[0744] C.3.2 置换多项式的逆

在本节中，我们给出计算$Z/(2^n)[x]$中任何给定置换多项式的合成逆的以下算法。

[0745] 命题 2。令$f(x)=\sum_{i=0}^{n+1}a_ix^{(i)}$是在$Z/(2^n)$上的置换多项式，并且令$f^{-1}(x)=\sum_{i=0}^{n+1}b_ix^{(i)}$是其置换逆。以下步骤提供用于计算$f^{-1}(x)$的系数。

[0746] 1. 输入$f(x)$；

2. 对于从 0 到(n+1)的 i

(a) 输入$f(x)$和 i 到命题??；

(b) 输出x_i(注意：$f(x_i)=i$)；

3. $b_0=x_0$；

4. 对于从 1 到 n+1 的 j

$$(a)\ t_1=\sum_{k=0}^{j-1}b_kj^{(k)}$$
$$(b)\ t_2=(j!>>v(j))^{-1}mod2^n$$
$$(c)\ b_j=((x_j-t_1)>>v(j))*t_2mod2^n$$

5. 输出$b_0,b_1,\cdots,b_{n+1}$。

[0747] 算法的正确性基于以下论证。因为$f^{-1}(x)=\sum_{i=0}^{n+1}b_ix^{(i)}$由(n+2)对值$(i,f^{-1}(i))=(i,x_i)$确定，所以$f^{-1}(x)$的系数可以通过求解等式系统来计算。

[0748] 该算法的复杂度为$O(n^2)$。

[0749] C.4 多项式的乘法逆函数

对于在$Z/(2^n)$上的给定多项式函数$f(x)$，我们希望确定$f(x)$是否具有乘法逆函数$g(x)$使得对于所有$x\in Z/(2^n)$有$f(x)*g(x)=1$，并且如果其存在的话由$f(x)^{-1}$来表示。我们还希望计算$f(x)^{-1}$。

[0750] 令$MIP(2^n)[x]$是$Z/(2^n)[x]$中的所有乘法可逆多项式的集合。

[0751] C.4.1 $f(x) \in \mathrm{MIP}(2^n)[x]$ 的准则

命题 3。令 $f(x)$ 是在 $Z/(2^n)$ 上的多项式函数。

[0752] 1. 如果且仅如果在递降阶乘幂表达式中系数 c_0 是奇数并且 c_1 是偶数，$f(x)$ 具有乘法逆函数 $f(x)^{-1}$ ；

2. 在 $Z/(2^n)[x]$ 中存在 $2^{n(n+2)-\sum_{k=1}^{n+1}\tau(k)-2}$ 乘法可逆多项式 ；

3. $f(x)^{-1}$ 是多项式并且可以通过有效的算法来计算。

[0753] 证明 ：明显地，在 $Z/(2^n)$ 上，如果且仅如果对于所有 $x \in Z/(2^n)$，$f(x)\&1 = 1$，$f(x)$ 具有乘法逆函数。在其递降阶乘幂表达式中，仅 $x^{(1)}$ 的系数和常数在最低有效比特中扮演角色，因为对于所有 $k \geq 2$，2 除 $x^{(k)}$。如果 $x = 0$，c_0 必须是奇数，并且如果 $x = 1$，c_1 必须是偶数。另一方面，以这些条件，$f(x)\&1 = 1$ 对于所有 $x \in Z/(2^n)$ 为真。

[0754] 在以下命题中陈述有效算法。

[0755] 命题 4。令 $f(x)$ 是在 $Z/(2^n)[x]$ 中的乘法可逆多项式。其乘法逆 $f(x)^{-1}$ 可以通过以下步骤生成 ：

1. 设置

$$x_0 = f(x) + 8 * g(x),$$

其中 $g(x) \in Z/(2^n)[x]$ 是任意多项式 ；

2. 执行递归等式

$$x_{k+1} = x_k(2 - f(x) * x_k)$$

$\ln(n)$ 次以生成新多项式 $t(x)$ ；

3. 将 $t(x)$ 标准化为次数最多为（n+1）的递降阶乘表示 ；

4. 输出 $t(x)$。算法的准确性基于以下观察 ：对于任何可逆元素 $a \in Z/(2^n)$，在过程中使用的牛顿迭代使在计算 a^{-1} 的准确性方面的比特数量翻倍。使用数 8 是因为 $f(x)$ 和 $f(x)^{-1}$ 的前 3 个比特对于所有 x 是相同的，这归因于以下事实 ：对于所有奇数 $a \in Z/(2^n)$，a 和 a^{-1} 的前 3 个比特是相同的。

[0756] 因为多项式在合成操作下是闭合的，所以我们具有多项式格式的逆。

[0757] 注意，具有不同初始值的算法产生不同的中间计算，并因此产生多样化的代码。

[0758] 该算法的性能是高效的，因为其仅取 $\ln(n)$ 次迭代。该符号化计算产生多项式逆的方程，其可以用于计算逆的系数实例。

[0759] C.4.2 用于计算 $f(x)^{-1}$ 的算法

用于计算给定乘法可逆多项式 $f(x)$ 的逆的方法归因于以下事实 ：在 $Z/(2^n)$ 上的任何

多项式$f(x)$可以通过值$\{f(0),f(1),\cdots,f(n+1)\}$的集合来确定。以下是简单算法。

[0760] 1. 取乘法可逆$f(x)$作为输入；

2. 计算$\{f(0),f(1),\cdots,f(n+1)\}$；

3. 计算模逆$\{f(0)^{-1},f(1)^{-1},\cdots,f(n+1)^{-1}\}$；

4. 计算多项式$g(x)=f(x)^{-1}$的系数，

(a)基于值集合$\{g(0)=f(0)^{-1},g(1)=f(1)^{-1},\cdots,g(n+1)=f(n+1)^{-1}\}$来计算$g(x)$的递降阶乘格式系数$\{a_i\}$

(b)通过模数$2^{n-v(i)}$来修整系数a_i

(c)将递降阶乘格式转换成正常模式；

5. 输出$g(x)=f(x)^{-1}$。

[0761] 该算法由于以下简单事实而保持：$f(i)*g(i)\equiv 1 mod(2^n),i=0,1,\cdots,n+1$。修整系数的步骤是必要的，以便产生零系数以具有有效计算所需的最短表示。

[0762] C.4.3 具有幂零系数的乘法可逆多项式

在$Z/(2^n)$上的所有乘法可逆多项式形成组，$Z/(2^n)[x]$的单元组。其子组可以针对在减小数量的非零系数方面的有效计算而被研究。例如，幂零系数多项式的逆仍然是幂零系数多项式？如果是，这可以是有效的子集。以下结果是我们预期的。

[0763] 引理1。令具有幂零系数的$f(x)=a_0+a_1x+\cdots a_mx^m\in Z/(2^n)[x]$：$a_i^2\equiv 0 mod(2^n),i=1,2,\cdots,m$。于是：

1. 对于任何$s\in N$，$\partial(f(x)^s)\leq\partial f(x)$；

2. 如果$f(x)$是乘法可逆的，即如果a_0是奇数，则我们具有$\partial(f(x)^{-1})\leq\partial f(x)$；

3. 对于任何整数$m\in N$，集合

$$N_m(Z/(2^n))=\{\sum_{i=0}^{m}a_ix^i\in Z/(2^n)[x]|(a_0\&1)=1,a_i^2\equiv 0 mod(2^n),1\leq i\leq m\}$$ 是单元组$U((Z/(2^n))[x],*)$的子组。

[0764] 这里是简短证明。如果我们令$t(x)=f(x)-a_0$，则$t(x)^2\equiv 0$。因此，$f(x)^2=c_0+c_1t(x)$。类似地，$f(x)^3=d_0+d_1t(x)$。第一结果是从$s\in N$上的归纳产生的。第二结果可以通过牛顿迭代过程$x_{n+1}=x_n(2-f(x)*x_n)$（$x_0=1$）和第一结果来证明。事实上，$x_1=x_0(2-f(x)*x_0)=2-f(x)$并且$x_2=x_1(2-f(x)*x_1)$。通过归纳，$x_k$是$f(x)$的多项式并且具有不大于$f(x)$的次数的次数。第三结果是容易检查的，这再次归因于系数的幂零属性，并且证据是完整的。

[0765] 在前面子节中的算法的Smalltalk实现为我们提供在$Z/(2^{32})[x]$上的幂零系数多

项式的逆的示例：

1. 二次多项式 $f(x)=83248235+1726834042470 4*x+2342188220116*x^2$，逆 $f(x)^{-1}=1416251459+2857631744*x240386048*x^2$，其也是二次多项式；

2. 三次多项式 $f(x)=1235235789+6887876591616*x4678345031680*x^2+1393796343398 4*x^3$，逆 $f(x)^{-1}=646443269+3893362688*x+2192048128*x^2+1208221696*x^3$，其也是三次多项式；

3. 四次多项式 $f(x)=98653231+1720291426304*x233922192299840*x^2+1393677070761984*x^3+1393816790630 4*x^4$，逆 $f(x)^{-1}=2846913231+3760455680*x+3063152640*x^2+1806172 16*x^3+200540160*x^4$，其也是四次多项式。

[0766] 注解 1。幂零系数条件可以通过$a_i^t\equiv mod(2^n)0.t\geq 2$而是松弛的。需要更详细的研究。

[0767] C.5 置换和乘法可逆多项式的分解、因式分解

在本节中，我们研究具有小表示的多项式$f(x)$。这是有用的，因为在$\mathbb{Z}/(2^n)$上的一般多项式具有次数$(n+1)$，并且如果高次置换用作变换，则变换后的代码将变得无效率。此外，代码混淆基于以下合理性也可以从小表示获益：小语言部件使代码多样性和一致性管理更容易。注意，在数据变换的上下文中，对于$f(x)$和其逆$f^{-1}(x)$(在$\mathrm{MIP}(2^n)$的情况中$f(x)^{-1}$)二者都是需要的，这被发现为是有挑战性的问题。

[0768] 对于给定置换多项式$f(x)$，其在常规多项式表示的非零项的数量被定义为其重量$wei(f(x))$(在递降阶乘表示中，我们可以具有较小的重量定义，但是其将被不同地处理，因为不存在重复的平方算法在这里工作)。显然，$wei(f(x))\leq deg(f(x))$。为了具有为小表示的$f(x)$和$f^{-1}(x)$二者，在次数上放置限制是明显的选项，已知提供一类置换多项式$f(x)$使得$deg(f(x))=deg(f^{-1}(x))$。另一方面，找到具有小$wei(f(x))$和$wei(f^{-1}(x))$的$f(x)$是找到有用的小表示的选项，因为存在有效的取幂计算，诸如重复的平方方法。

[0769] 多项式函数分解为我们提供了找到具有小表示的$f(x)$的另一手段。如果$f(x)=g(h(x))$，则$deg(g(x))$和$deg(h(x))$是$deg(f(x))$的整数因数。对于多变量多项式我们具有类似的情况，其可以是针对算术运算的变形器代码。

[0770] 多项式因式分解是我们具有小表示的第三种方法。注意，在$GF(2)[x]$中存在次数为 m 的约 1/m 个不可约多项式。

[0771] 例如，在$GF(2)$上仅存在 99 个次数为 10 的多项式是不可约的。幸运的是，置换多项式(远非不可约的。现有的数学规则使得在$GF(2)$上的$f(x)$

的任何因式分解外延到在$\mathbb{Z}/(2^n)$上的因式分解(非唯一),任何$n \geq 2$,并且在$GF(2)$上的用于作为置换(或乘法可逆)的多项式系数条件不将其限制为不可约的。在该上下文中,对于$f(x)$和$f^{-1}(x)$,理想的小表示是少量因数,并且每个因数是多项式与小表示的幂,所述小表示诸如低次数小表示或低重量的小表示(一种递归定义)。

[0772] 使用混合加法项、合成部件、乘法因数表示多项式提供另一极佳的示例:相同的技术组服务于有效计算和软件混淆目的二者(现有示例是用于RSA密钥混淆的加法链)。因为我们的最终目标是用于变形器代码,所述变形器代码是三个置换多项式与算术运算以及双变量多项式的合成,作为最后结果,我们具有好机会来构造/找到小表示的大量置换多项式,以具有优化后的结果代码(一般算法应变确定——我们具有一些基本思想)。

[0773] 在以下子小节中,我们描述用于这三种方法以及它们的混合的算法。

[0774] C.5.1 低次数或低重量$f(x)$

我们已经获得了关于置换多项式$f(x)$的足够条件使得$f(x)$和$f^{-1}(x)$二者具有相同的次数,所述次数可以小。

[0775] 这里是关于$h(f(x)+g(y))$的次数的结果,其中$f(x),g(y),h(z) \in P_m(\mathbb{Z}/(2^n))$。

[0776] 令m是正整数并且令$P_m(\mathbb{Z}/(2^n))$是在$\mathbb{Z}/(2^n)$上的多项式的集合:

$$P_m(\mathbb{Z}/(2^n)) = \left\{ \sum_{i=0}^{m} a_i x^i \;\middle|\; \forall a_i \in \mathbb{Z}/(2^n), a_1 \wedge 1 = 1, a_i^2 = 0, i = 2, \ldots, m \right\}。$$

[0777] 次数是m并且存在2m-1个系数。

[0778] 我们研究了较少限制条件以及对于$deg(f(x)) = deg(f^{-1}(x))$可能必要且充分的条件的情况,但是结果是基于系数等式的系统的理论条件是复杂的,但是其确实在这样的多项式的计算上照射了一些光(细节在这里被省略)。在该点处,我们使用计算来进行搜索,并且如果计算结果可以提供进一步的信息则将恢复理论研究。

[0779] 基本计算算法要应用于在节C.3.2中的算法以用于逆的计算,并调谐系数以找到小表示。

[0780] 在伽罗瓦域上,诸如低重量不可约多项式的低重量多项式通过计算来研究。在该伽罗瓦环$\mathbb{Z}/(2^n)$的情况中,我们应当再次使用节C.3.2中的算法,并找到第重量的那些。系数调谐过程在运行时再次发生。

[0781] 5.2 分解

在域(不一定是有限的)上的多项式时间的分解方法是已知的,但是在伽罗瓦环上,就我们迄今为止的知识而言,还未找到/发现令人信服的一般算法。另一方面,在域上的方法和思想提供用于在环上工作的有价值的信息。

[0782] 特殊类别的称为Chebyshev多项式的多项式(第一种类的多项

式）$T_n(x)$ 值得我们注意。回顾 $T_n(x)$ 可以通过以下递归关系来定义：$T_0(x)=1, T_1(x)=x, T_{n+1}(x)=2xT_n(x)-T_{n-1}(x)$。Chebyshev 多项式的属性是关于分解：$T_{nm}(x)=T_n(T_m(x))$。令人感兴趣的观察是所有奇数索引的多项式 $T_{2k+1}(x), k=1,2,\ldots,$ 是在 $\mathbf{Z}/(2^n)$ 上的置换多项式。因此，大奇数索引的 Chebyshev 多项式可以分解成低次 Chebyshev 置换多项式。

[0783] 注意，如果 $f(x)=g(h(x))$ 并且 $g(x)$ 和 $h(x)\in \mathbf{Z}/(2^n)[x]$，则那些分量 $g(x)$ 和 $h(x)$ 仍然是置换。对乘法可逆的分解将是令人感兴趣的，因为分量不一定是乘法可逆的。

[0784] C.5.3 $f(x)$ 和 $f(x,y)$ 的因式分解

给定多项式 $f(x)\in \mathbf{Z}/(2^n)[x]$ 的因式分解在 $\mathbf{Z}/(2)[x]=GF(2)[x]$ 处开始。然后，各种形式的 Hensel 提升可以被选择来在 $\mathbf{Z}/(2^n)$ 上对 $f(x)$ 因式分解。在该领域中的算法得到了很好地研究（除了对多变量多项式进行因式分解以外），并且我们将使用现有的算法。

[0785] 大多数置换多项式不是基本本原多项式并且具有非平凡因数。例如，置换多项式 $f(x)=x+8*x^2+16*x^3=x(4*x+1)^2$。

[0786] 对于任何 $f(x)\in \mathbf{Z}/(2^n)[x]$，无平方因式分解算法和 Berlekamp 的 Q 矩阵算法用于对 $f(x)\in \mathbf{Z}/(2)[x]$ 进行因式分解。注意，我们可能仅具有部分因式分解，找到互质因数，以去往下一步骤来对 $f(x)\in \mathbf{Z}/(2^i)[x], i\geq 2$ 因式分解。

[0787] 以下形式的 Hensel 的引理是具有技术实质的引理。

[0788] 令 R 是环并且理想 $I\subset R$。对于任何 $f\in R$ 以及在 R/I 中的 f 的使得 $gcd(g,h)\equiv 1(mod I)$ 的任何因式分解 $f\equiv gh(mod I)$，存在 g^* 和 h^* 使得 $f\equiv g^*h^* mod I^2$，$g^*\equiv g mod I$，$h^*\equiv h mod I$，以及此外，$gcd(g^*,h^*)\equiv 1 mod I^2$。

[0789] 还注意，g^* 和 h^* 可以直接从 $gcd(g,h)\equiv 1(mod I)$ 的 Bzout 恒等式构造。

[0790] 迭代该过程，我们可以具有期望结果。

[0791] 注意，置换多项式的因数不一定是置换。这在不同种类间的多样性方面提供了另一特点。然而，乘法可逆多项式的因数仍然是乘法可逆的。

[0792] C.5.4 混合加法项、乘法因数以及合成分量

我们知道在 $\mathbf{Z}/(2^n)$ 上的所有置换多项式基于函数合成运算 $\circ$ 而形成组 $PP(2^n)[x]$。基于环乘法的多项式环 $\mathbf{Z}/(2^n)[x]$ 的单元组 $U(\mathbf{Z}/(2^n)[x],\cdot)$ 是所有乘法可逆多项式 $MIP(2^n)[x]$ 的集合（参见节 C.4）。

[0793] 这里是简单但令人感兴趣的观察：

命题 5。令 $f(x)=xh(x)\in PP(2^n)[x]$，具有零常数项的置换多项式。于是 $h(x)\in MIP(2^n)[x]$。即，$h(x)$ 是乘法可逆的。

[0794] 注意，$f(x)$ 的 $x^{(1)}, x^{(2)}$ 和 $x^{(3)}$ 的系数必须分别是奇数、偶数和偶数。以该格式，这

些条件使$h(x)$的常数项能够是奇数，并且$x^{(1)}$的系数是偶数。该观察的正确性遵循命题 3。

[0795] 另一观察是交$\mathsf{PP}(2^n)[x] \cap \mathsf{MIP}(2^n)[x]$，其为空（如果我们允许$\mathsf{PP}(2^n)[x]$具有常数函数，则仅包含奇数常数函数）。这暗指两个函数集合在某种意义上是正交的。

[0796] 回到（第一种类的）Chebyshev 多项式$T_n(x)$的集合。之前我们提到奇数索引的那些是置换。容易看到偶数索引的$T_{2k}(x)$（也是偶数索引的第二种类的 Chebyshev 多项式）是乘法可逆多项式。因此，大偶数索引的多项式可以基于$T_{nm}(x) = T_n(T_m(x))$被分解成小的那些，并且可替代地，其可以被因式分解成用于可约的那些的小因数。

[0797] 在该领域中可以完成更多研究，包括出于生成高度混淆的代码的目的选择合适变换的算法。

[0798] C.6 一般化的多项式$\widetilde{f(x)} = a_0 \overset{\oplus}{+} a_1 x \overset{\oplus}{+} \cdots \overset{\oplus}{+} a_d x^d$

对于作为置换的一般化的多项式$\widetilde{f(x)}$，如果且仅如果条件基于其系数集合$\{a_0, \cdots, a_d\}$，则 Klimov 给出兴趣。这是用于通过用 + 替换所有$\oplus$运算来获得的多项式函数$f(x) = \sum_{i=0}^{d} a_i x^i$的相同条件，称为$\widetilde{f(x)}$的约简多项式。

[0799] 因为联合定律在$\widetilde{f(x)}$中是无效的，所以它实际上表示从所有可能定序的运算以及运算符 + 和$\oplus$的组合生成的函数的集合。

[0800] 令人感兴趣的关于单个周期属性的条件的新结果在 2011 年末发布：假设运算顺序是从左到右，即，$\widetilde{f(x)}$具有格式为$\{\cdots((a_0 \overset{\oplus}{+} (a_1 x)) \overset{\oplus}{+} (a_2 x^2)) \overset{\oplus}{+} \cdots \overset{\oplus}{+} (a_d x^d)\}$的函数。

[0801] 命题 6。在定序限制和假设不存在连续 + 运算符的情况下$\widetilde{f(x)} = a_0 \overset{\oplus}{+} a_1 x \overset{\oplus}{+} \cdots \overset{\oplus}{+} a_d x^d$是单个周期置换，如果且仅如果其是在环$\mathbb{Z}/(2^{2^l+2^t})$上的单个周期置换，其中$l$是在$\{i_1, i_2, \cdots, i_m\} \subset \{1, 2, \cdots, d\}$中的奇数的数量，其是在它们之前具有 + 运算符的项$a_{s_j} x^{s_j}$的次数索引的集合。

[0802] 这是令人感兴趣的结果，但是$\mathbb{Z}/(2^{2^l+2^t})$可能是大环。

[0803] C.7 矩阵变换

在$\mathbb{Z}/(2^n)$上具有预定行列式函数的矩阵函数在本节中构造，以用于算术运算和矢量的变换。以下是我们尝试工作于的矩阵的集合：

$$\Omega = \{M_{s\times s}(a_{i,j}(x,y,z,\cdots)) \mid s \in \mathbb{N}, a_{i,j}(x,y,z,\cdots) \in \mathsf{B}^n[x,y,z,\cdots],$$
$$|M| = g(x), \forall g(x) \in \mathsf{MIP}(2^n)\}$$

其中$\mathsf{B}^n[x,y,z,\cdots]$是在 BA[n] 上的多变量函数。

[0804] 回顾$\mathsf{MIP}(2^n)$是在$\mathbb{Z}/(2^n)$上的所有乘法可逆多项式的集合。

[0805] 几条解释线。这是其行列式是乘法可逆矩阵的矩阵集合。以预定行列式，在Ω中的矩阵可以基于在环$\mathbb{Z}_f(2^n)$上的基本行和列运算来构造。注意，也涉及在BA[n]中的其他运算，但是我们“关心”乘法和加法。以下与在域上的矩阵非常类似的标准算法提供更多细节。

[0806] 命题7。令$m,n\in\mathbb{N}$。令$\mathbb{A}_n$是来自称为上下文函数集合的$\mathbf{BA}[n]\{x,y,z,\cdots\}$的函数集合。令$\mathbb{A}_n^{\infty}$是从$\mathbb{A}_n$生成的函数的集合。以下过程生成在$\mathbf{BA}[n]$上的可逆矩阵$M=(m_{i,j}(x,y,z,\cdots))_{m\times m}$（更精确地，条目$m_{i,j}(x,y,z,\cdots)\in(\mathbb{A}_n\cup MIP(2^n))^{\infty}$），其行列式是多项式$f(x)\in MIP(2^n)$：

1. 输入：BA代数维度$n,\mathbb{A}_n,MIP(2^n)$和矩阵维度m；

2. 从$MIP(2^n)$和集合$m_{i,j}(x,y,z,\cdots)=f_i(x)$随机选择m个多项式$f_1(x),f_2(x),\cdots,f_m(x)$；

3. 重复以下过程的有限步骤：

(a)随机拾取$i,j\in\{1,2,\cdots,m\}\times\{1,2,\cdots,m\}$；

(b)随机拾取$r(x,y,z,\cdots)\in\mathbb{A}_n^{\infty}$；

(c)随机执行行运算$R_i*r(x,y,z,\cdots)+R_j$或列运算$C_i*r(x,y,z,\cdots)+C_j$；

4. 输出：具有维度$\prod_{i=1}^{m}f_i(x)$的$m\times m$矩阵。

[0807] 在算法中，上下文函数集合$\mathbb{A}_n$是令人感兴趣的概念。来自$\mathbf{BA}[n]\{x,y,z,\cdots\}$的该函数集合定义矩阵变换所需的“相似度”以无缝掺合到应用代码环境中。$\mathbb{A}$可以基于现有的应用代码格式和射影代码格式来预先定义。$\mathbb{A}$的典型示例是在代码上下文中的表达式的集合。

[0808] 该概念还可以帮助我们在应用和变换二者中在代码变量之间引入相关性。参见示例节D。

[0809] 注解2。替代算法通过以下来构造该矩阵：具有$\mathbf{MIP}(2^n)$中的多项式的上(或下)三角矩阵形成来自$\mathbb{A}_n^{\infty}$的对角条目和元素以形成上(或下)条目。这些矩阵的乘积仍然在$\mathbf{MIP}(2^n)$中。

[0810] 注解3。关于在Ω中的矩阵的逆的一致性。存在两种类型的应用：在应用代码中有或没有矩阵逆代码。为了变换矢量(数据阵列)，其可能不是必要的，因为逆计算可以在服务器侧发生，这不需要代码混淆。但是使用矩阵逆来变换运算中的操作数称为必要的以保持原始功能性。

[0811] 在Ω中的矩阵良好地服务于后一种情况，因为逆矩阵的条目由来自$\mathbb{A}_n$和多项式(行列式的逆)构成。

[0812] 注解4。算法可以基于定义的代码一致性级别的精确准则和变换后的代码的执行准则而被微调。

[0813] C.8 块可逆函数矩阵

在本节中，我们利用块可逆属性来构造平方块矩阵的特殊集合，所述块可逆属性要应用于内部和外部变换二者都保持乘法可逆性的代码上下文。

[0814] 注意，这也是在白盒 AES 密钥隐藏中应用的常数情况构造的外延。

[0815] C.8.1 块可逆函数矩阵的存在

在本节中，我们将$(\mathbf{Z}/(2^n))[x]$中的偶数多项式称为其 x 的系数和常数项二者都是偶数的多项式(与多项式环$(\mathbf{Z}/(2^n))[x]$的幂零根有关)。令子集$\Psi \subset (\mathbf{Z}/(2^n))[x]$是乘法可逆多项式与偶数多项式的并。于是

$$\Psi = \{f(x) = \sum_{i=0}^{n+1} c_i x^i \in (\mathbf{Z}/(2^n))[x] | c_i \text{ is even}\}$$

是子环$(\Psi, +, *)$。令在环Ψ中生成的思想 Let $\Xi = < 2(\mathbf{Z}/(2^n))x, (\mathbf{Z}/(2^n))x^i | i = 2, 3, \cdots, n+1 >$。

[0816] 容易验证Ψ/Ξ与 Z 同构：乘法可逆多项式的集合成为奇数，并且偶数多项式变为偶数。我们将看到该同构性将在[?]中的域$\mathbf{Z}/(2)$上的构造方法变换成环Ψ。

[0817] 注意，Ψ包含由单元组$U((\mathbf{Z}/(2^n))[x], *)$和幂零根思想$N((\mathbf{Z}/(2^n))[x], *)$生成的子环。矩阵环$M(\Psi)_{*X*}$是我们在本节中工作于的内容。首先，我们具有以下结果：

引理 2。对于给定非零平方矩阵$A \in M(\Psi)_{*X*}$，存在两个可逆矩阵$P, Q \in M(\Psi)_{*X*}$使得$M = P * D * Q$，其中 D 是具有 r 个一和 s-r 个零的对角矩阵，其中$r \in N$。

[0818] 引理 3。对于任何$s, r \in N$（$s \geq r$），存在两个可逆矩阵$T, A \in M(\Psi)_{*X*}$使得$T = D + A$，其中 D 是具有 r 个一和 s-r 个零的对角矩阵，其中$r \in N$。

[0819] 这两个引理的正确性遵循以上同构性。对于Ψ中的该多项式子集，构造算法的基本思想在函数矩阵情况中在此工作得很好(即使在 BA 代数上，实质上在模数环$\mathbf{Z}/(2^n)$上)。

[0820] 10 程序部件的一般变换

我们描述用于变换程序部件的方法(参见下一节，用于作为主要变换的算术运算和数据阵列及置换多项式、可逆多项式以及矩阵)。注意，在本节中描述的方法实质上是数据变换概念 IRDETO/Cloakware 数据变换技术的复杂度外延。

[0821] 10.1 变换过程和配置

配置的定义。在该注释中，配置被定义为在变换过程中涉及的变量集合(无运算)的状态。在大多数情况中，变量集合包括：

1. 程序部件中的变量；
2. 程序部件的变换后的变量；
3. 它们的类型表示的变换；
4. 变换的系数和 / 或它们的值；
5. 变换的系数和 / 或它们的逆的值；
6. 表示配置自身的变量；

7. 其他变量。

[0822] 变换过程包括一系列：

1. 输入配置；

2. 部件变换部分；

3. 输出配置。

[0823] 作为变换过程的主要部分的配置管理可以通过有限状态机来表示。注意，程序的变换(加密)比用于密码目的的二进制串更加复杂。

[0824] 示例，现有数据变换。

[0825] 10.2 对变换后的部件和程序的复杂度分析

1. 搜索所有可能合成的空间；

2. 变换后的程序部件的相关性的传播的复杂度；

3. 用于每个实例的等式系统。

[0826] 11 在标记 I/ 木人构造中的程序部件的变换

基于木人构造要求，我们具有以下程序部件变换。

[0827] 11.1 加法的变换

令$z = x + y$是要变换的加法。基本过程是：

1. 对输入配置解码以找出 x 和 y 二者的输入表达式；

2. 通过在$PP(2^n)$中的置换多项式来变换两个操作数并生成新的表达式；

3. 用经编码的操作数和作为其条目的其他项来创建矢量；

4. 通过Ω中的矩阵来变换该矢量；

5. 通过$PP(2^n)$中的多项式和 / 或Ω中的矩阵来合成对矩阵和两个操作数的解码与对加法运算的编码；

6. 向变换 z 应用置换多项式并将结果保存在矢量中；

7. 为加法的消耗者将关于编码的信息保存在最后矢量中。

[0828] 那些步骤之间的接口是具有指定 / 不同配置的变量阵列。

[0829] 11.2 乘法的变换

类似于以上步骤，仅仅用乘法替换加法。

[0830] 11.3 矢量 / 阵列的变换

类似于加法步骤，没有加法编码。

[0831] 11.4 加法、乘法和矢量的变换

通过矩阵变换的选择来统一这三种变换。

[0832] 13 攻击

对于置换多项式——简化的表示。

[0833] 命题 8。可能的攻击：如果所有置换多项式可以由低次置换多项式表示，则存在(简化)攻击 HINT：对在有限域上的低次置换的数量进行计数以表面这些不是这样的攻击。

[0834] 命题9。可能的攻击：使用值集合$f(i)|i \in 0,1,\cdots,n+1$来表示多项式和合成以计算概率……。

[0835] 14 注释

算法的 Java 代码实现被测试。

[0836] 15 动态置换变换的概念

在 Irdeto/Cloakware 中的现有数据变换技术中，置换用于变换运算的操作数。例如，加法运算 $w = u + v$ 的操作数 u、v 和 w 可以通过线性函数 $y = a * x + b$ 进行变换，其中 a 和 b 是常数(必须是作为置换的奇数)。在变换后的代码的计算中，变量与 u 和 v 相关，仅因为变换后的运算具有作为新变换后的运算的系数的固定常数，诸如用于加法运算的那些。因此，我们将称这些变换为静态置换变换。注意，在诸如 RSA 的标准公钥密码系统中使用的数据变换也可以认为是静态置换变换，因为私钥和公钥对在实体使用它们对数据加密或解密时成为固定常数。

[0837] 相对地，动态置换变换是具有新变量的置换，所述新变量可以引入到变换的计算上下文中。例如，线性函数 $y = a * x + b$ 可以用于变换操作数 x。但是在该动态情况中，系数 a 和 b 是变量(对变量 a 有一比特限制)。在该方式，变换后的加法运算将具有三个集合的新变量(在该情况中总共 6 个)。

[0838] 静态置换变换被设计为变换每个单独运算的操作数。这些宏变换始终具有小代码大小。尽管动态置换变换可以用作宏变换，但是其注意目标是在用于代码完整性的变量之间引入连接。因此，代码大小可以并且应当比大多数宏变换更大。注意，这些大大小的变换仍然在关于原始大小的多项式时间计算复杂度的边界内。

[0839] 动态和静态置换变换可以一起工作来以合理的代码大小扩展实现代码混淆和代码保护的级别。

[0840] 16 动态置换变换

在 $\mathbb{Z}/(2^n)$ 上的置换多项式 $f(x) = y_0 + y_1 * x + \ldots + y_n * x^n$ 可以用作动态置换变换，其中 $y_0, y_1, \cdots, y_n$ 是具有以下条件的变量：y_1 是奇数，$y_2 + y_4 + \cdots$ 和 $y_3 + y_5 + \cdots$ 是奇数。如在静态数据变换情况中那样，置换逆必须被计算。

[0841] 除了在 $\mathbb{Z}/(2^n)$ 上的一般置换多项式以外，诸如具有幂零系数的那些的特殊动态置换多项式可以减小变换后的代码的大小。计算它们的逆的方程在[？]中是已知的，其中所有系数是该动态情况中的变量。诸如幂零属性的系数变量的特殊属性也可以用于互锁。

[0842] 注意，没有什么会防止动态置换变换中的系数变量具有一些常数变量，只要置换条件被满足即可。它们将促进与现有静态置换变换的合成。

[0843] 17 动态置换变换和互锁的属性

在动态置换变换中，存在两种系数变量：条件变量，诸如在 $y = a * x + b$ 中的 a，和非条件变量，诸如在以上示例中的 b。出于代码混淆目的，非条件系数变量可以是来自原始计算上下文的任何变量，或者在任何变换方程中的变量。条件变量是更令人感兴趣的：条件可以与用于代码保护目的的互锁属性进行合成。代码可以被这样保护是因为系数变量的条件确切地是用作作为置换的变换的条件。因此，破坏条件暗示对于运算的操作数的非置换变换，而导致错误计算，这是我们想到其会在篡改出现时发生的方式。

[0844] 因为动态置换条件由一组变量的属性来表示，所以变得难以从原始代码属性中区分这些属性。也难以从在变换后的代码中的所有变量算出系数变量集合。

[0845] 除了系数变量的属性以外，方程的正确性的条件也可以与完整性验证属性合成：

如果条件被破坏，则其将破坏$f(g(x))=x$恒等式！

置换多项式也完全由其根来确定。除了正常系数表示以外，也可以使用根表示格式。特殊的根结构和值属性也可以减小代码大小以用于更高效的计算。注意，在该动态上下文中，根是变量而非固定值。针对根计算过程的正确性的条件也可以与验证属性合成。

[0846] 在计算逆的过程中的其他动态属性也可以用于与完整性属性合成。例如，用于计算在环$\mathbb{Z}/(2^n)$上的模数逆的基于牛顿迭代算法的算法仅针对奇数值正确工作，这是个好属性。

[0847] 17.1 恒等式和动态等式

固有地涉及多个变量的等式具有动态属性：恒等式自身。混合的布尔算术恒等式是那些等式的示例：MBA 恒等式的破坏暗示篡改的发生。

[0848] 在环$\mathbb{Z}/(2^n)$上的乘法可逆动态多项式$f(x)$还提供一组等式$f(x)*f(x)^{-1}=1$。类似于动态置换多项式，关于系数变量的条件也提供用于与完整性验证属性合成的属性。具有特殊系数变量的多项式提供柔性代码大小的实现。

[0849] 动态置换变换与 MBA 恒等式合成。这可以通过变换等式中的变量或者变换等式自身来完成。

[0850] 17.2 置换 T 函数

一般而言，来自布尔算术代数系统的任何置换 T 函数也可以用作动态置换变换。对它们的逆的计算可以通过逐比特计算来实现。一个示例是一般化置换多项式。对它们的逆的计算仍然是高效的计算机程序。

[0851] 注意：不是所有程序部件都是对于混淆必要的。不是数据变量的变换而是完整性代码仅与变量和运算合成。

[0852] 18 块数据保护

诸如在类中的数据结构或字段中的成员的块数据变量可以(1)使用动态置换多项式来变换：系数可以是单独数据变量；(2)与每个数据变量的单独静态置换变换合成。

[0853] 参考文献

1. G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*, Oxford Press.
2. Zhaopeng Dai, and Zhuojun Liu, *The Single Cycle T-functions*, On line: **http://eprint.iacr.org/2011/547.pdf**
3. Alexander Klimov, *Applications of T-functions in Cryptography*, PhD Thesis, Weizmann Institute of Science, 2004.
4. Dexter Kozen, Susan Landau, *Polynomial decomposition algorithms*, J. Symb. Comp, 7(5)(1989),445-456.
5. A. Menezes, P. Oorschot, S. Vanstone, *Handbook of Applied cryptography*, CRC Press, 1996.
6. Madhu Sudan, *Algebra and computation*, MIT lecture notes. On line: **http://people.csail.mit.edu/madhu/FT98/course.html**
7. G. Mullen, H. Stevens, Polynomial functions (mod m), Acta Math. Hungary, 44(3-4)(1984), 237-241.
8. Ronald L. Rivest, Permutation Polynomials Modulo 2^w, Finite Fields and their Applications, vol. 7, 2001, pp 287–292.

9. Henry S. Warren, Jr., *Hacker's Delight*, Addison-Wesley, Boston, 2002. On line: www.hackersdelight.org.
10. James Xiao, Y. Zhou, *Generating large non-singular matrices over an arbitrary field with block of full rank*, 2002. On line: http://eprint.iacr.org/2002/096.pdf.
11. Kejian Xu, Zhaopeng Dai and Zongduo Dai *The formulas of coefficients of sum and product of p-adic integers with applications to Witt vectors*, Acta Arithmetica. 150 (2011), 361-384. On line: http://arxiv.org/abs/1007.0878 http://journals.impan.pl/cgi-bin/doi?aa150-4-3.
12. Y. Zhou, S. Chow, System and method of hiding cryptographic private keys, U.S. Patent No. 7,634,091, 2009.
13. Y. Zhou, A. Main, Y. Gu and H. Johnson, Information Hiding in Software with Mixed Boolean-Arithmetic Transforms, *Information Security Applications, 8th International Workshop, WISA 2007*, LNCS 4867, 2008.

D.2 维矢量的变换的示例

这里是关于 2 维矢量 $(X,Y)\in(\mathbb{B}^{32})^2$ 的变换的非常简单的示例。我们假设代码上下文是：

$A_{32}=\{(x\oplus y),y^2,z,(x\&y)\}$。

[0854] 第一个步骤是拾取两个置换多项式 $39*x+42, 67*x+981$ 以变换 X 和 Y：

$$X_1=39*X+42,$$

和

$Y_1=67*Y+981$。

[0855] 下一个步骤是拾取其行列式是乘法可逆多项式 $(1335+2860*x+1657*x*(x-1))(6539+8268*x)$ 的矩阵：

$$A=\begin{pmatrix}(1335+2860*x+1657*x*(x-1)) & (67*(x\oplus y)+8*y^2)\\ 0 & (6539+8268*x)\end{pmatrix}。$$

[0856] 对 A 的列运算

$$C=\begin{pmatrix}1 & 0\\ 716*z+93*(x\&y) & 1\end{pmatrix}$$

为我们给出：

$A=A\times C=Matrix[[1335+2860*x+1657*x*(x-1)+(67*(x\oplus y)+8*y^2)*(716*z+93*(x\&y)),(67*(x\oplus y)+8*y^2)],[(6539+8268*x)*(716*z+93*(x\&y)),(6539+8268*x)]]$.

然后，对 A 的行运算

$$R=\begin{pmatrix}1 & 0\\ 34*(x\&y) & 1\end{pmatrix}$$

产生

$$A = R \times A = Matrix[[1335 + 2860 * x + 1657 * x * (x-1) + (67 * (x \oplus y) + 8 * y^2) * (716 * z + 93 * (x\&y)), 67 * (x \oplus y) + 8 * y^2], [34 * (x\&y) * (1335 + 2860 * x + 1657 * x * (x-1) + (67 * (x \oplus y) + 8 * y^2) * (716 * z + 93 * (x\&y))) + (6539 + 8268 * x) * (716 * z + 93 * (x\&y)), 34 * (x\&y) * (67 * (x \oplus y) + 8 * y^2) + 6539 + 8268 * x]].$$

向变换(X_1, Y_1)应用该可逆矩阵 A，我们具有变换后的矢量(X_2, Y_2)，其中：

$$\begin{aligned} X_2 = {} & 52065 * X + 56070 + 46917 * x * X + 50526 * x + 64623 * x^2 * X \\ & + 69594 * x^2 + 1870908 * (x \oplus y) * z * X + 2014824 * (x \oplus y) * z \\ & + 243099 * (x \oplus y) * (x\&y) * X + 261702 * (x \oplus y) * (x\&y) \\ & + 223392 * y^2 * z * X + 240576 * y^2 * z + 29016 * y^2 * (x\&y) * X \\ & + 31248 * y^2 * (x\&y) + 4489 * (x \oplus y) * Y + 65727 * (x \oplus y) \\ & + 536 * y^2 * Y + 7848 * y^2, \end{aligned}$$

并且

$$\begin{aligned} Y_2 = {} & 8110908 * x + 7595328 * (x\&y) * y^2 * z * X \\ & + 63610872 * (x\&y) * (x \oplus y) * z * X + 2234718 * (x \oplus y) * (x\&y) \\ & + 266832 * y^2 * (x\&y) + 182595036 * z * X + 553956 * x * Y \\ & + 1062432 * y^2 * (x\&y)^2 + 248635296 * x * z + 25487163 * (x\&y) * X \\ & + 34012692 * (x\&y) * x + 2366196 * (x\&y) * x^2 + 8897868 * (x \oplus y) * (x\&y)^2 \\ & + 8179584 * (x\&y) * y^2 * z + 68504016 * (x\&y) * (x \oplus y) * z \\ & + 18224 * y^2 * (x\&y) * Y + 152626 * (x \oplus y) * (x\&y) * Y + 230875632 * x * z * X \\ & + 986544 * y^2 * (x\&y)^2 * X + 8262306 * (x \oplus y) * (x\&y)^2 * X \\ & + 2197182 * (x\&y) * x^2 * X + 31583214 * (x\&y) * x * X \\ & + 6414759 + 27447714 * (x\&y) + 196640808 * z + 438113 * Y. \end{aligned}$$

然后，我们可以通过包括 X 和 Y 的任何变量的任何表达式或在代码上下文中的常数来替换 x、y 和 z，以向这些变换后的代码中注入新的相关性。进一步的代码优化取决于所选的表达式可以变得必要。

[0857] E. 携带比特值的多项式表示

给定两个数 a 和 b 的二进制表示 $a = \sum_{i=0}^{\infty} a_i 2^i$ 和 $b = \sum_{i=0}^{\infty} b_i 2^i$ 以及和 $c = a + b = \sum_{i=0}^{\infty} c_i 2^i$，用比特值$a_j, b_j, 0 \le j \le t$ 来表示携带比特是令人感兴趣的问题。最近，发展处了方程。

$$c_0 = a_0 \oplus b_0$$

[0858] 对于 $t \ge 1$，

$$c_t = (a_t \oplus b_t) \oplus (\oplus_{i=0}^{t-1}(a_i \cdot b_i \cdot (H_{j=i+1}^{t-1}(a_j \oplus b_j))))。$$

[0859] 明显地，以上第二项是携带比特值的多项式表示。也可以导出类似的用于乘法的

方程。

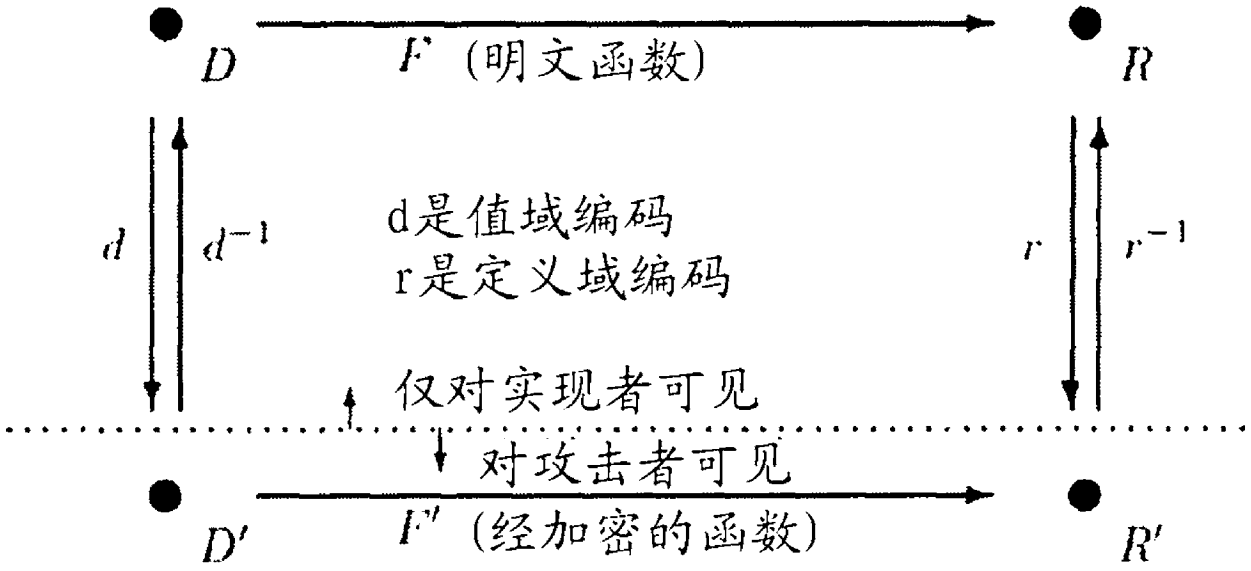

D
F（明文函数）
R
d
d^{-1}
d是值域编码
r是定义域编码
r
r^{-1}
仅对实现者可见
对攻击者可见
D'
F'（经加密的函数）
R'

图 1

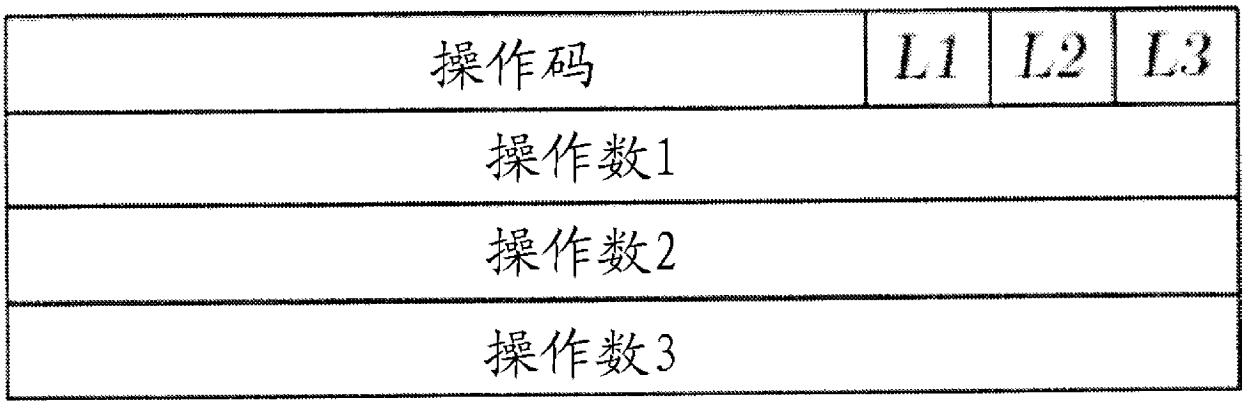

操作码
L1
L2
L3
操作数1
操作数2
操作数3

图 2

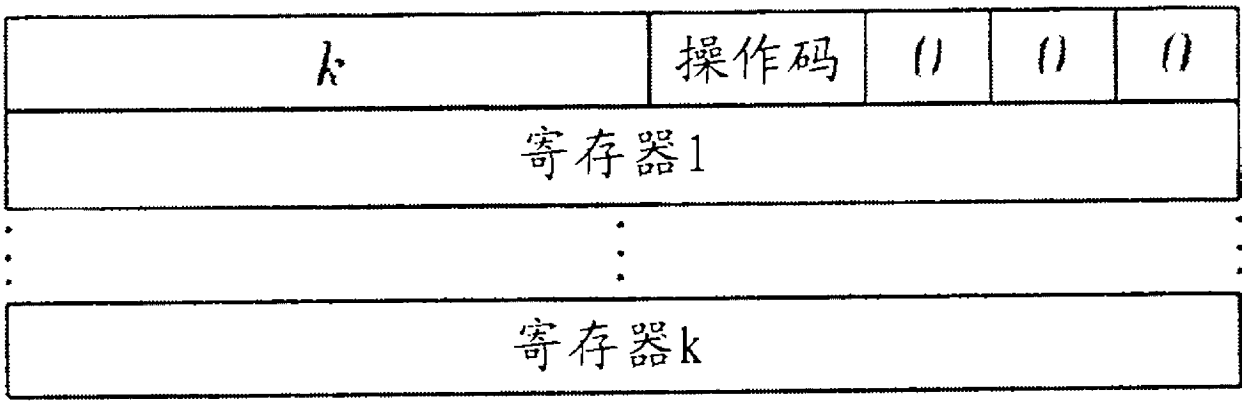

k
操作码
0
0
0
寄存器1
寄存器k

图 3

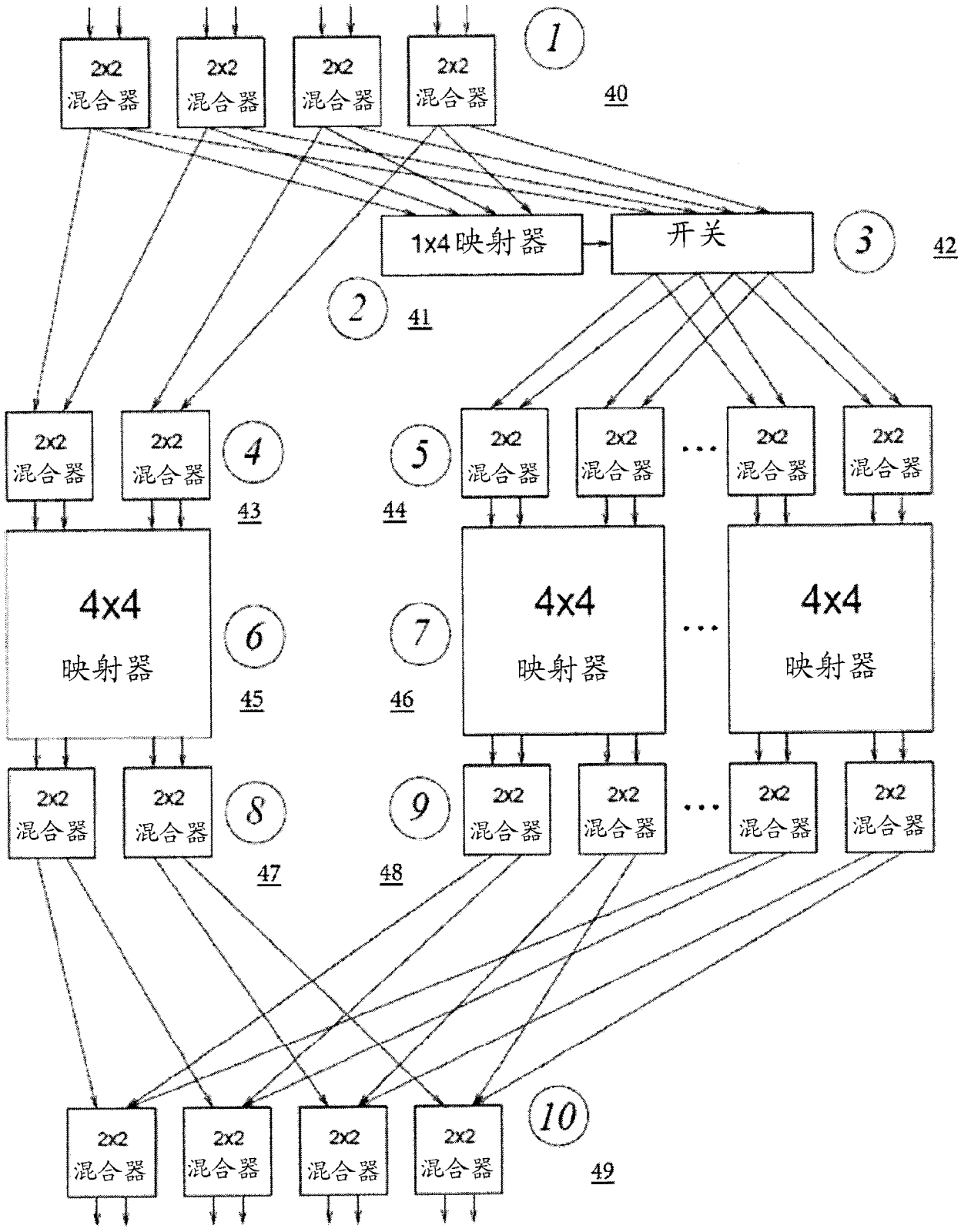

2x2 混合器
1
40
1x4映射器
开关
3
42
2
41
4
43
5
44
4x4 映射器
6
45
7
46
8
47
9
48
10
49

图 4

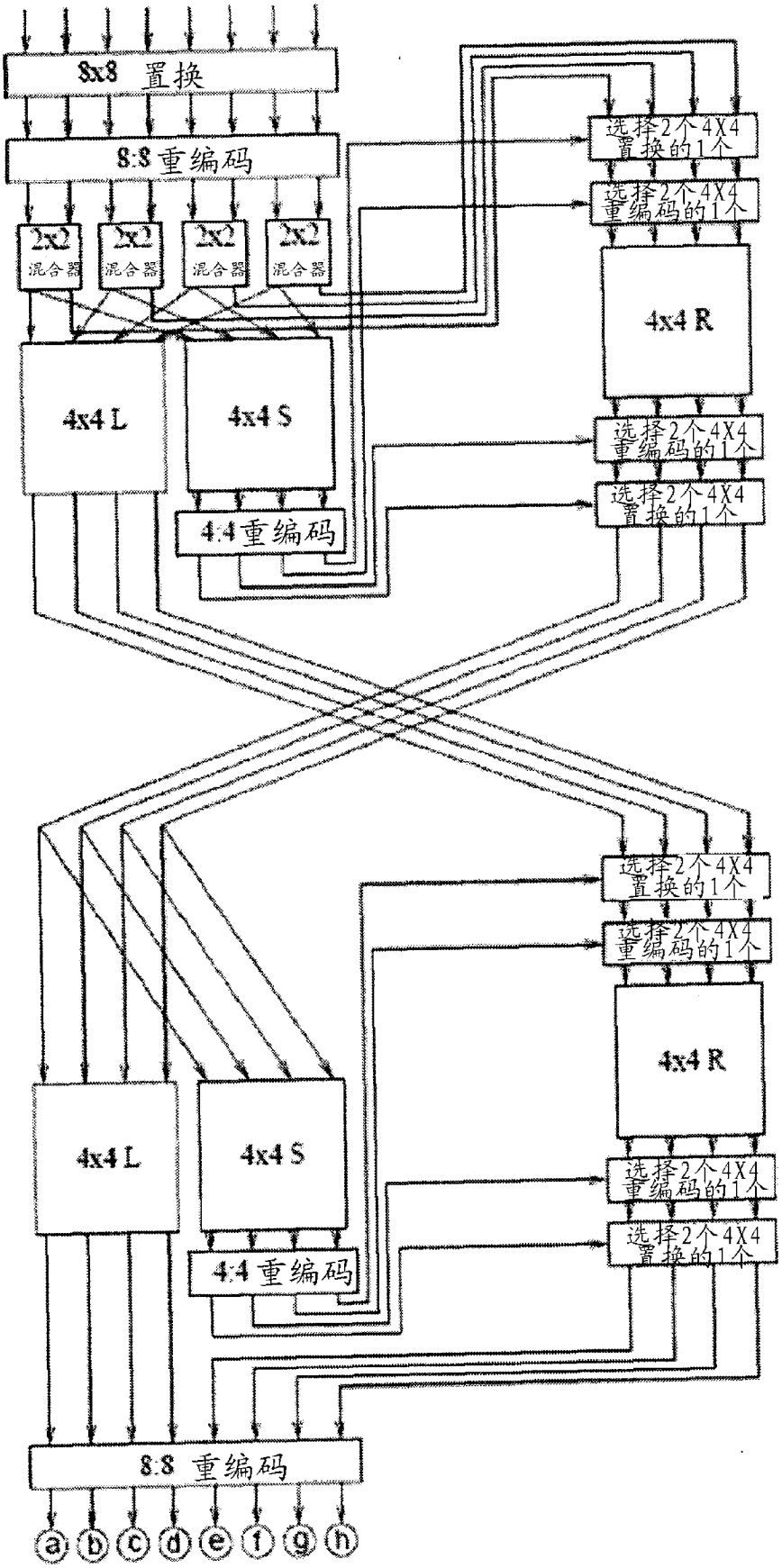

8x8 置换
8:8 重编码
2x2 混合器
2x2 混合器
2x2 混合器
2x2 混合器
4x4 L
4x4 S
4:4 重编码
选择2个4X4置换的1个
选择2个4X4重编码的1个
4x4 R
选择2个4X4重编码的1个
选择2个4X4置换的1个
选择2个4X4置换的1个
选择2个4X4重编码的1个
4x4 R
4x4 L
4x4 S
4:4 重编码
选择2个4X4重编码的1个
选择2个4X4置换的1个
8:8 重编码
a
b
c
d
e
f
g
h

图 5

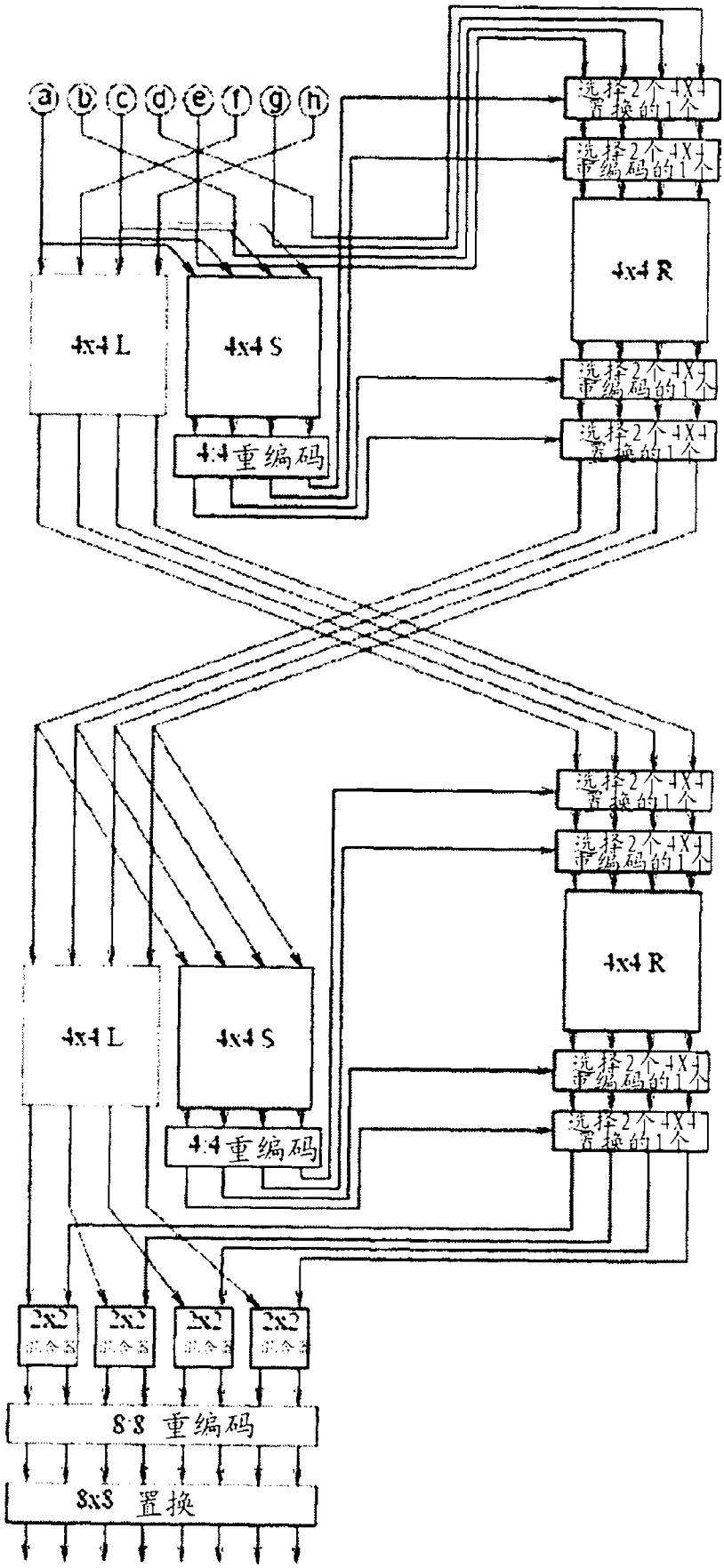

a
b
c
d
e
f
g
h
选择2个4X4置换的1个
选择2个4X4重编码的1个
4x4 R
4x4 L
4x4 S
4:4重编码
选择2个4X4重编码的1个
选择2个4X4置换的1个
选择2个4X4置换的1个
选择2个4X4重编码的1个
4x4 R
4x4 L
4x4 S
4:4重编码
选择2个4X4重编码的1个
选择2个4X4置换的1个
8:8 重编码
8x8 置换

图 6

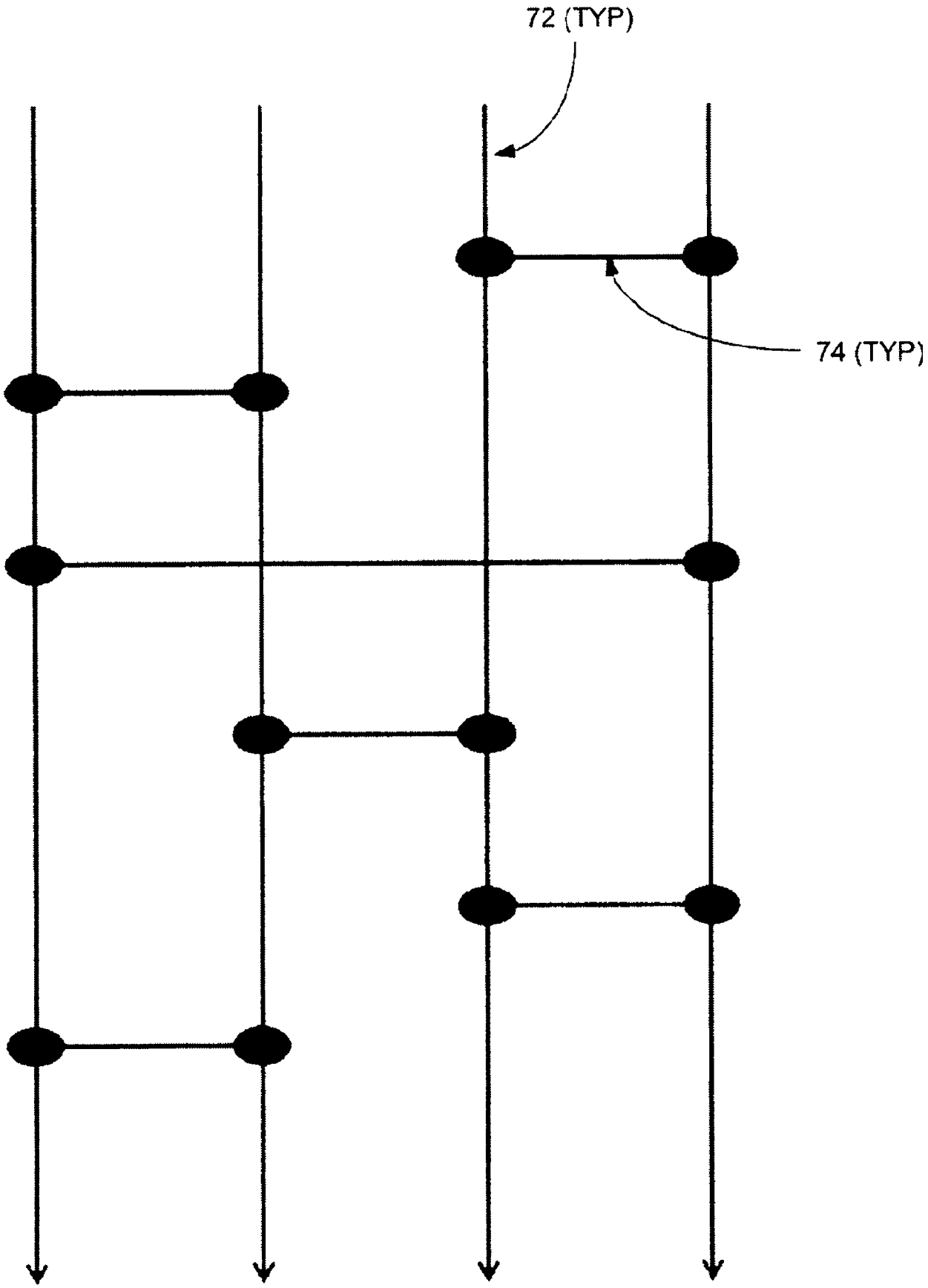


图 7

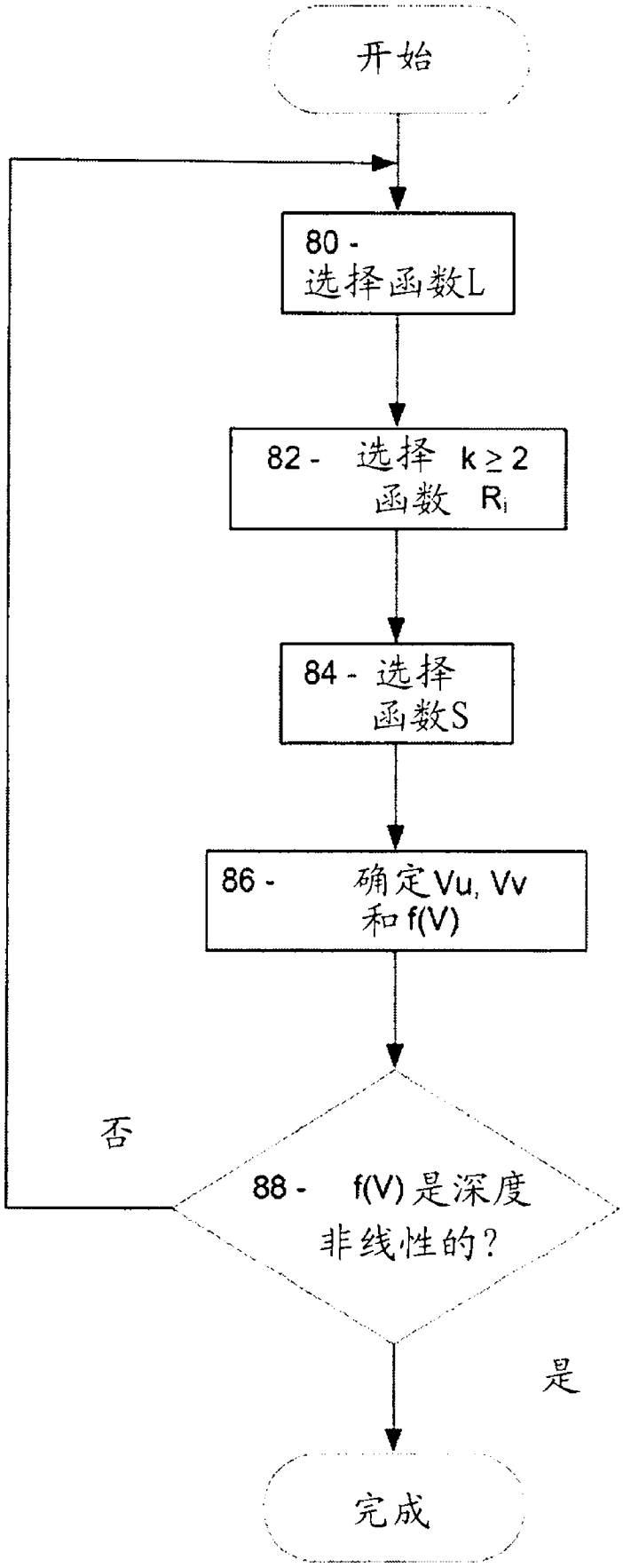

开始
80 - 选择函数L
82 - 选择 k ≥ 2 函数 R_i
84 - 选择函数S
86 - 确定Vu, Vv和f(V)
88 - f(V)是深度非线性的?
否
是
完成

图 8

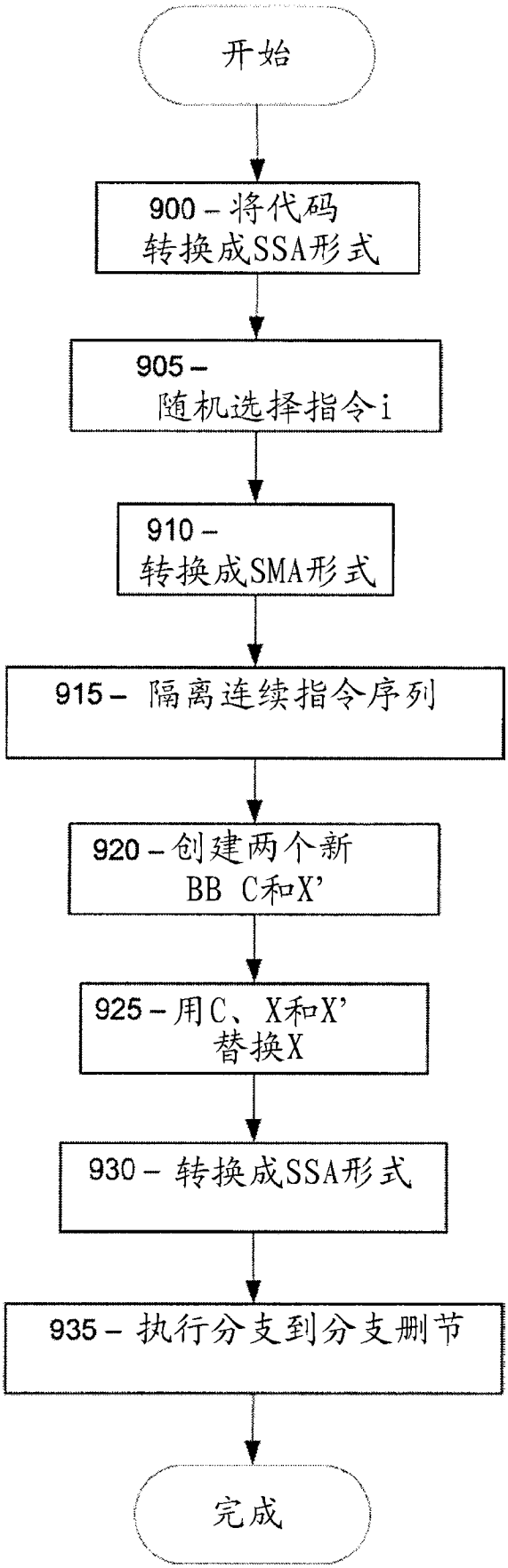

开始
900－将代码转换成SSA形式
905－随机选择指令i
910－转换成SMA形式
915－ 隔离连续指令序列
920－创建两个新BB C和X'
925－用C、X和X'替换X
930－ 转换成SSA形式
935－ 执行分支到分支删节
完成

图 9

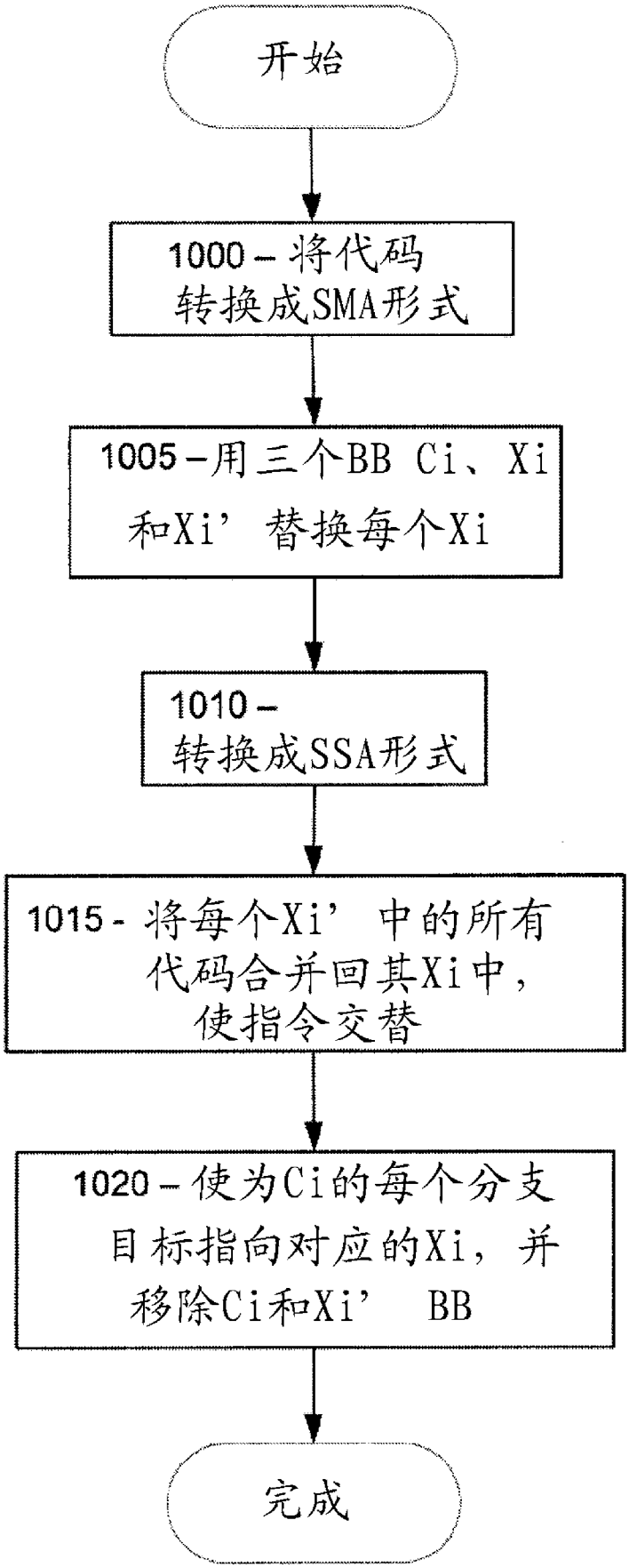

开始
1000－将代码转换成SMA形式
1005－用三个BB Ci、Xi和Xi’替换每个Xi
1010－转换成SSA形式
1015－将每个Xi’中的所有代码合并回其Xi中，使指令交替
1020－使为Ci的每个分支目标指向对应的Xi，并移除Ci和Xi’ BB
完成

图 10

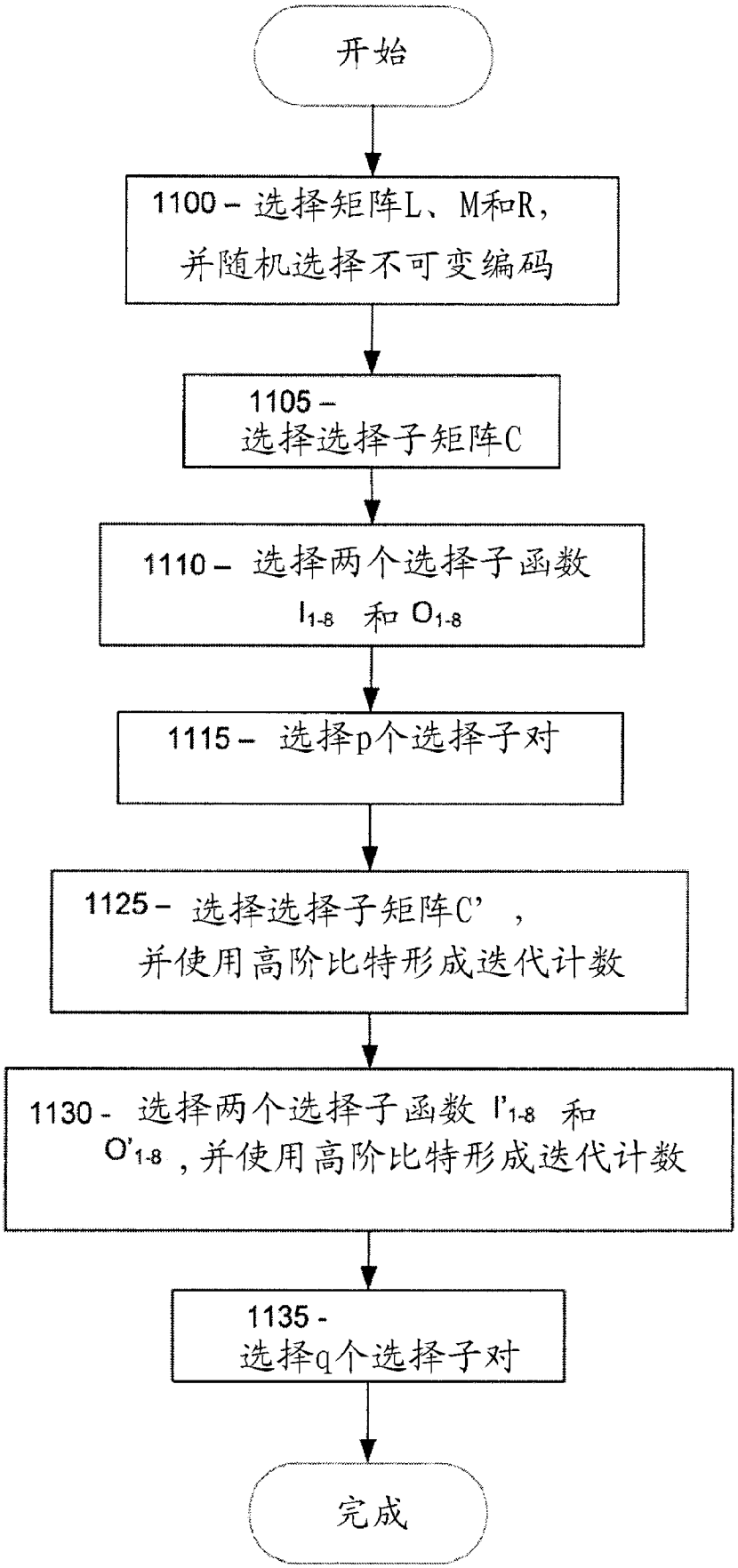

开始
1100 – 选择矩阵L、M和R，并随机选择不可变编码
1105 – 选择选择子矩阵C
1110 – 选择两个选择子函数 I_{1-8} 和 O_{1-8}
1115 – 选择p个选择子对
1125 – 选择选择子矩阵C’，并使用高阶比特形成迭代计数
1130 – 选择两个选择子函数 I'_{1-8} 和 O'_{1-8}，并使用高阶比特形成迭代计数
1135 – 选择q个选择子对
完成

图 11

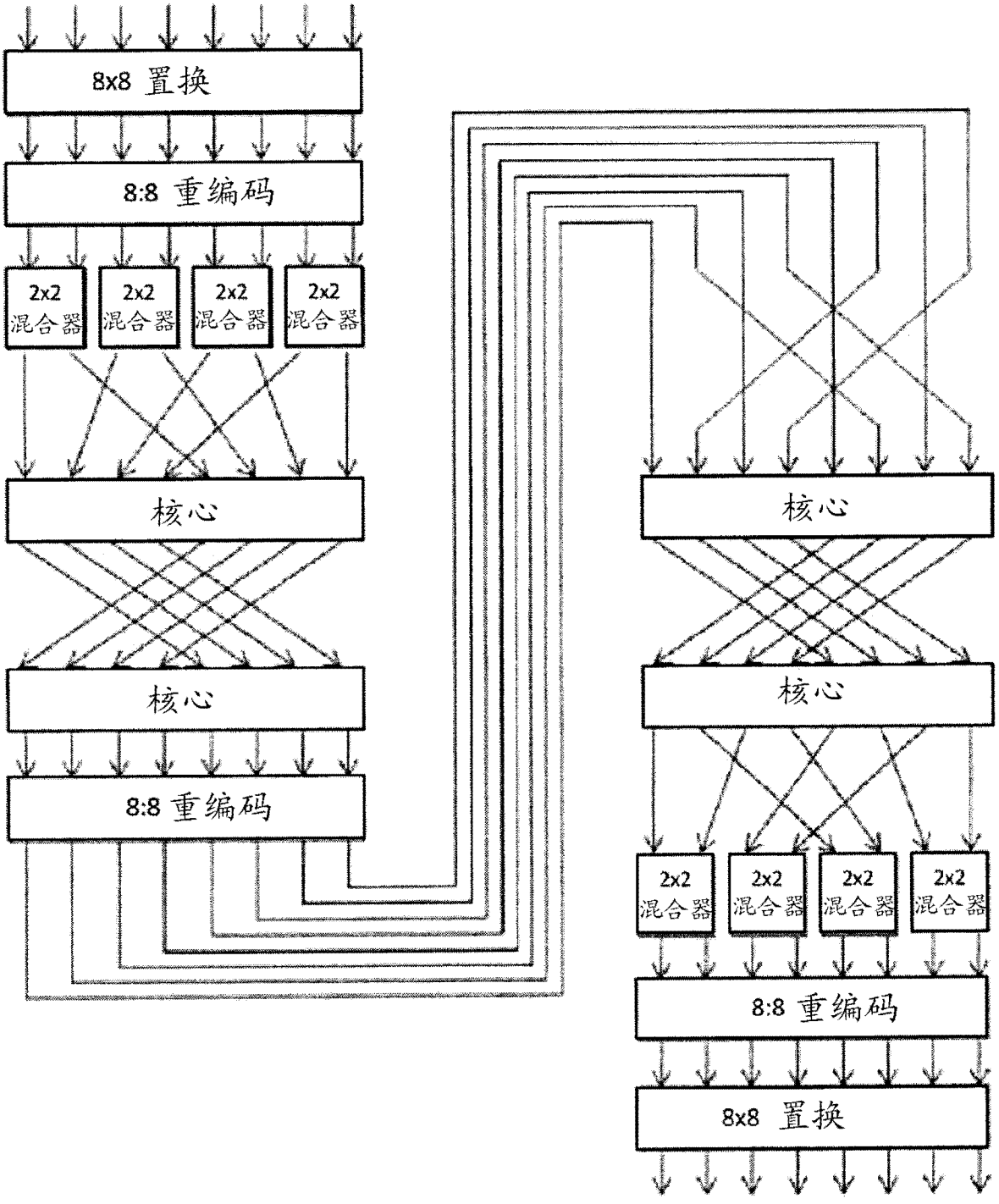

8x8 置换
8:8 重编码
2x2 混合器
2x2 混合器
2x2 混合器
2x2 混合器
核心
核心
8:8 重编码
核心
核心
2x2 混合器
2x2 混合器
2x2 混合器
2x2 混合器
8:8 重编码
8x8 置换

图 12

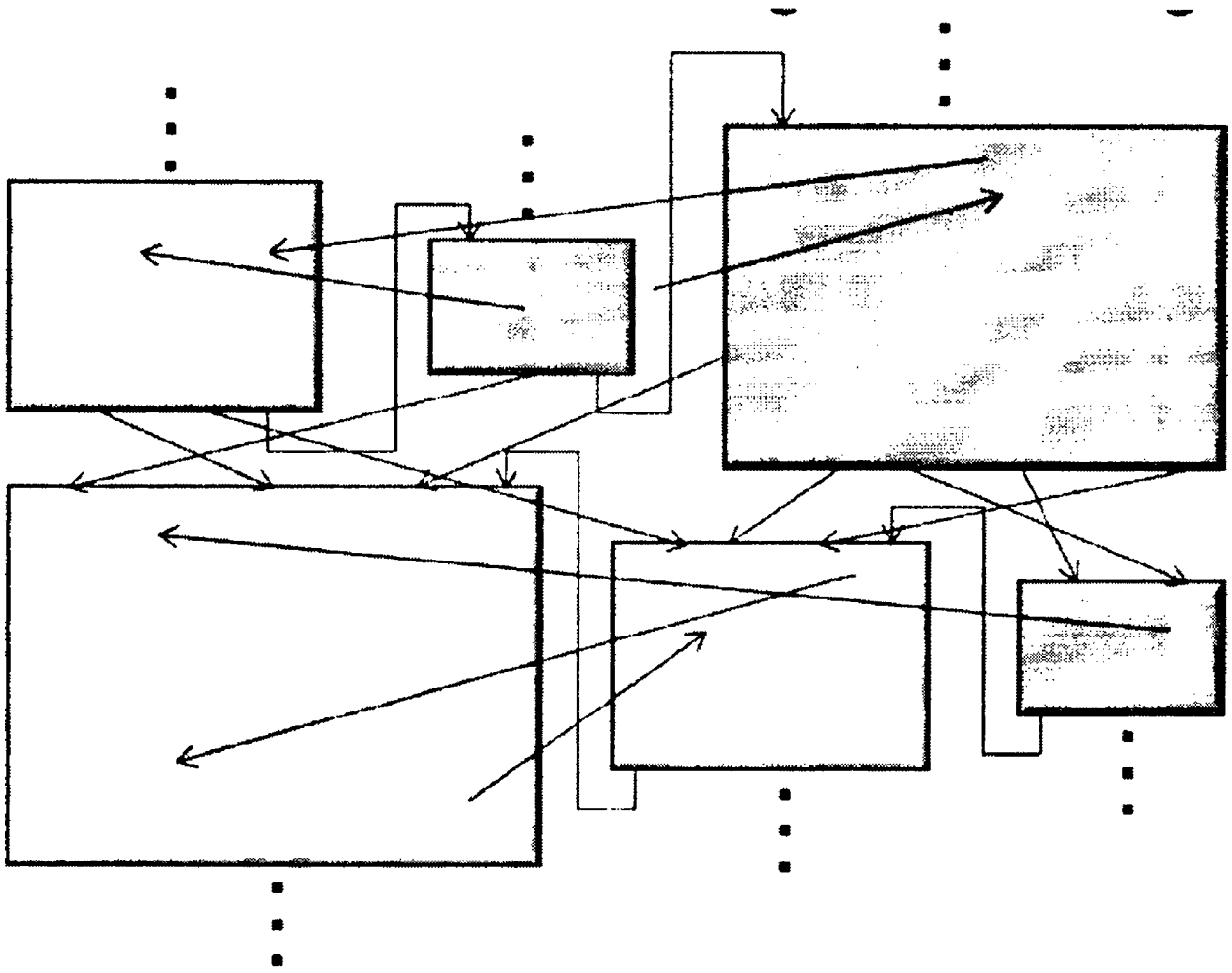


图 13

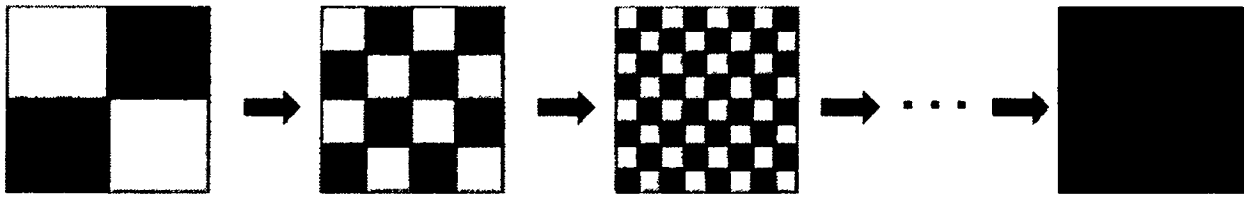


图 14

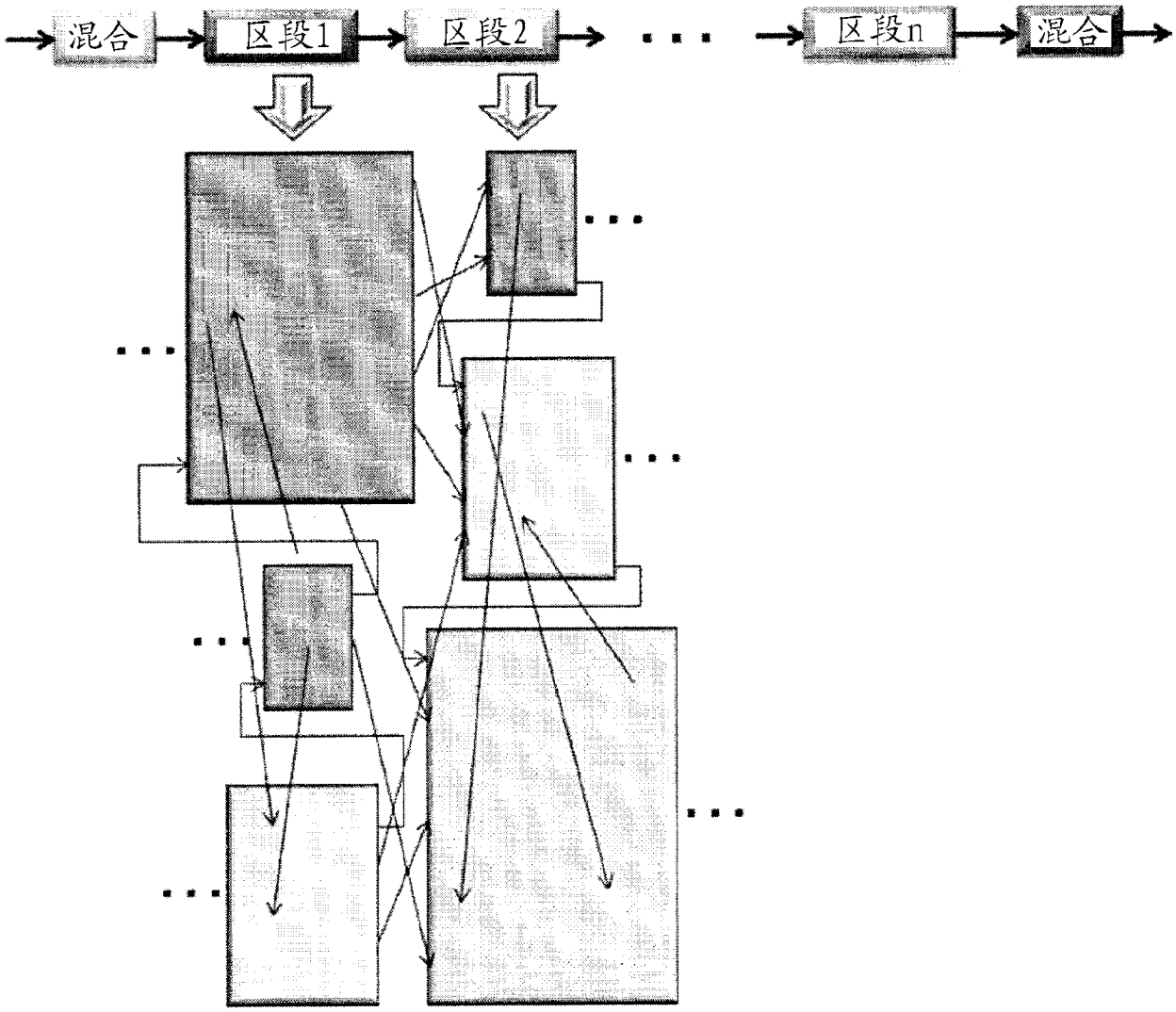


图 15

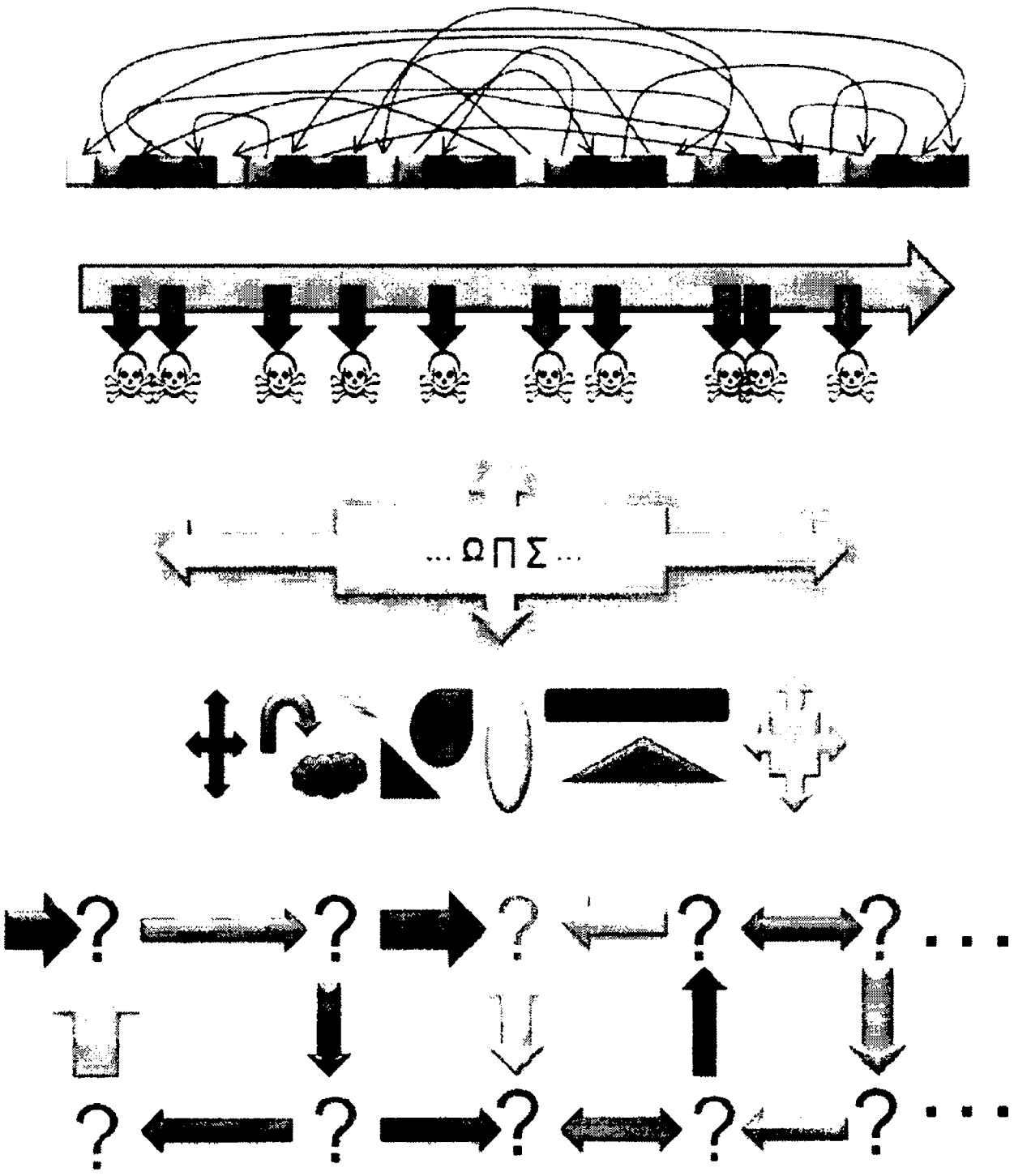


图 16

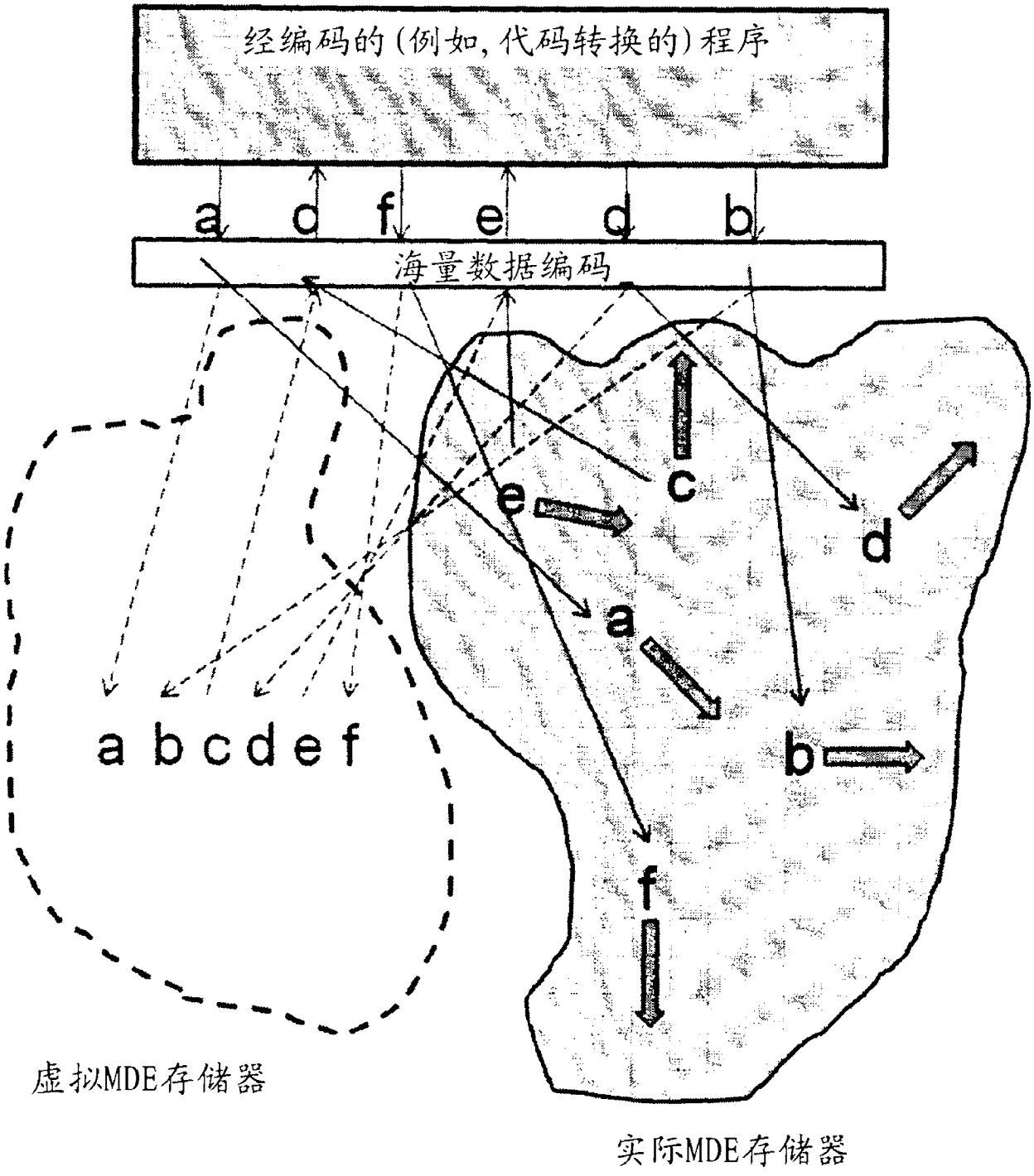

经编码的(例如,代码转换的)程序
a
c
f
e
d
b
海量数据编码
e
c
d
a
b
f
a b c d e f
虚拟MDE存储器
实际MDE存储器

图 17

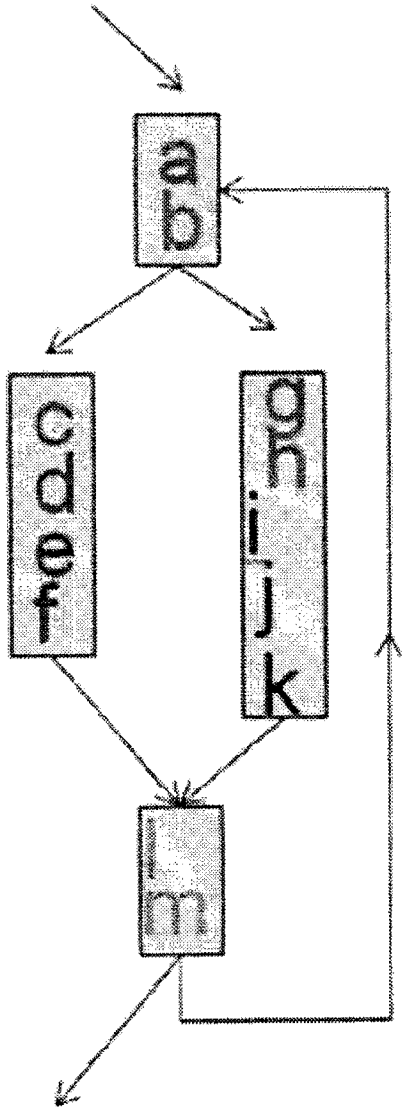


图 18

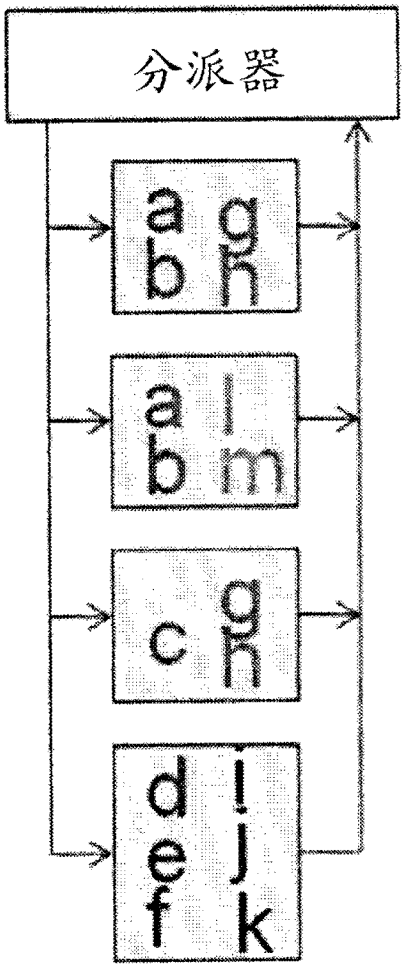


图 19

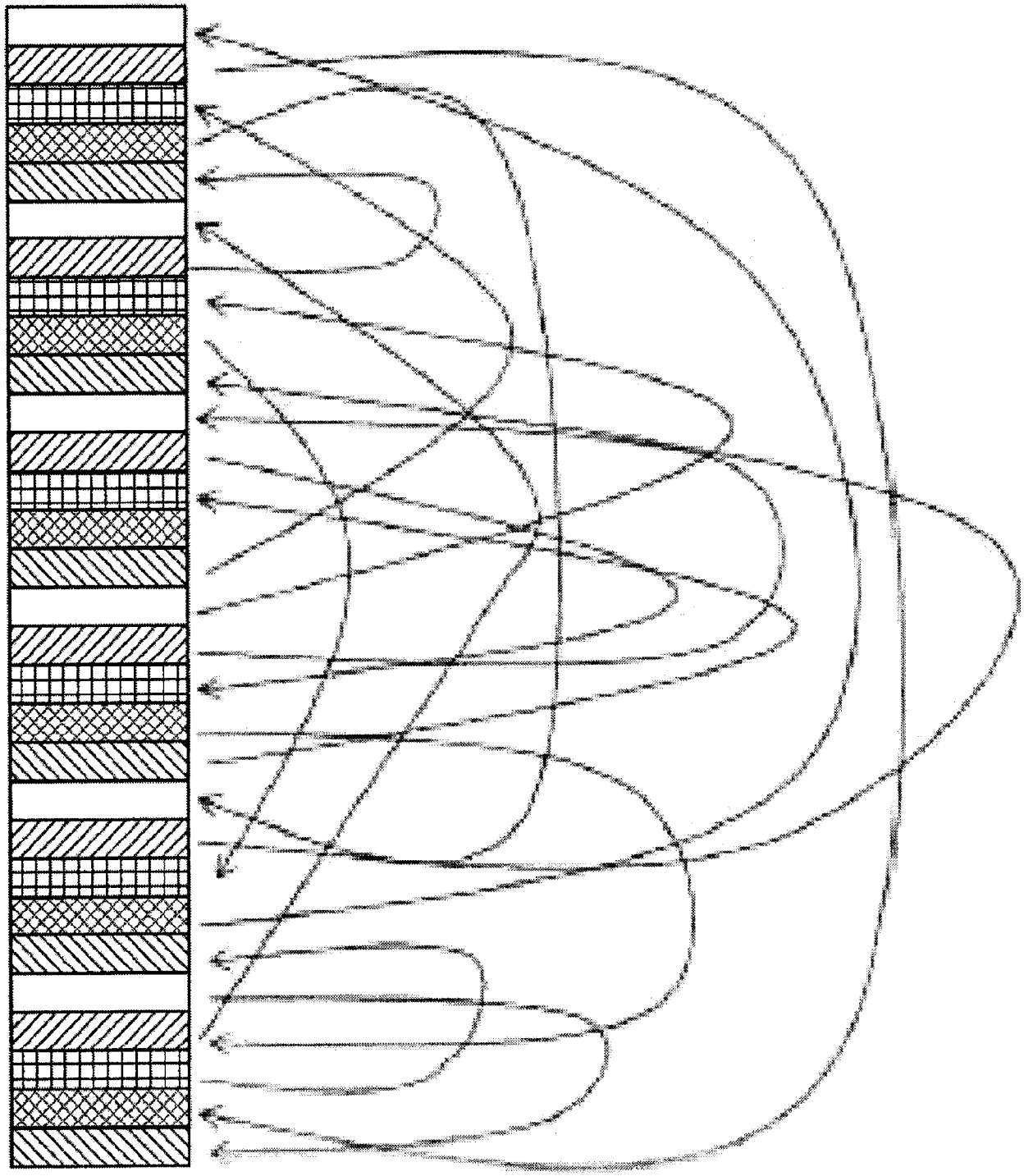
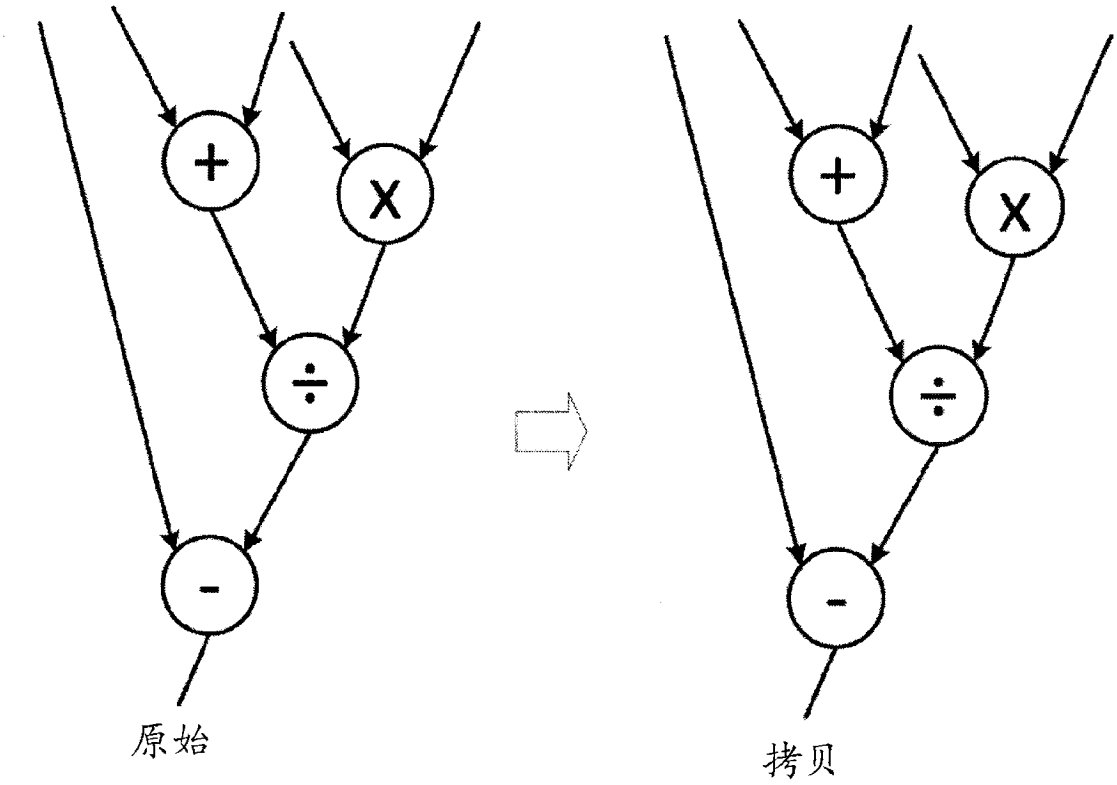


图 20



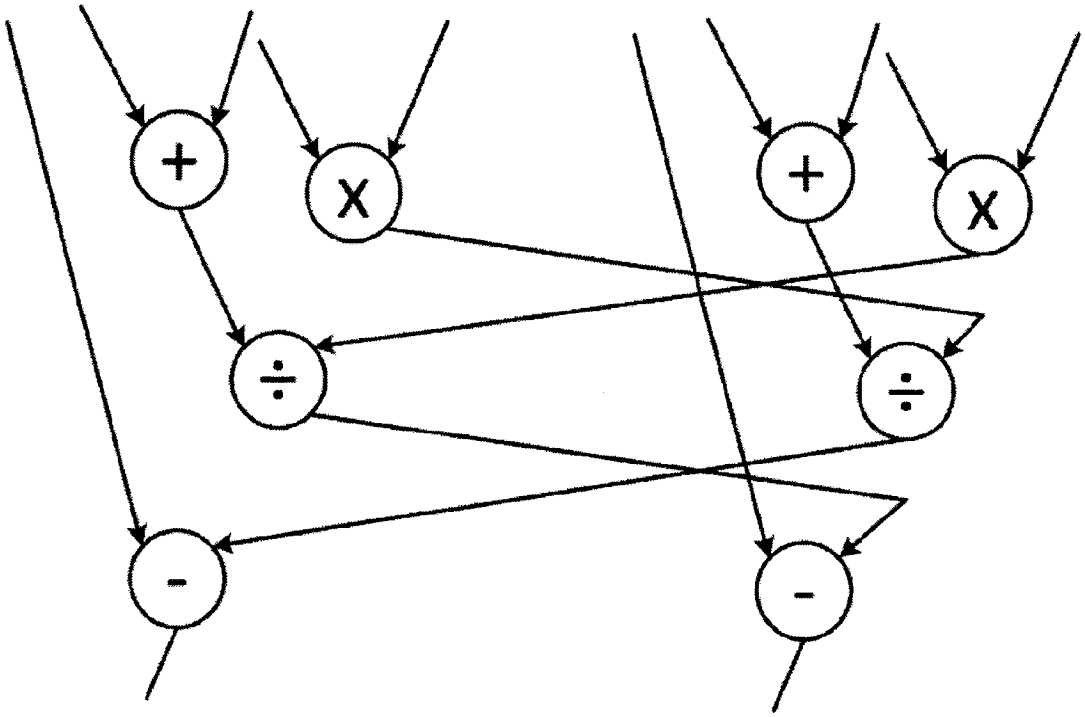


图 21

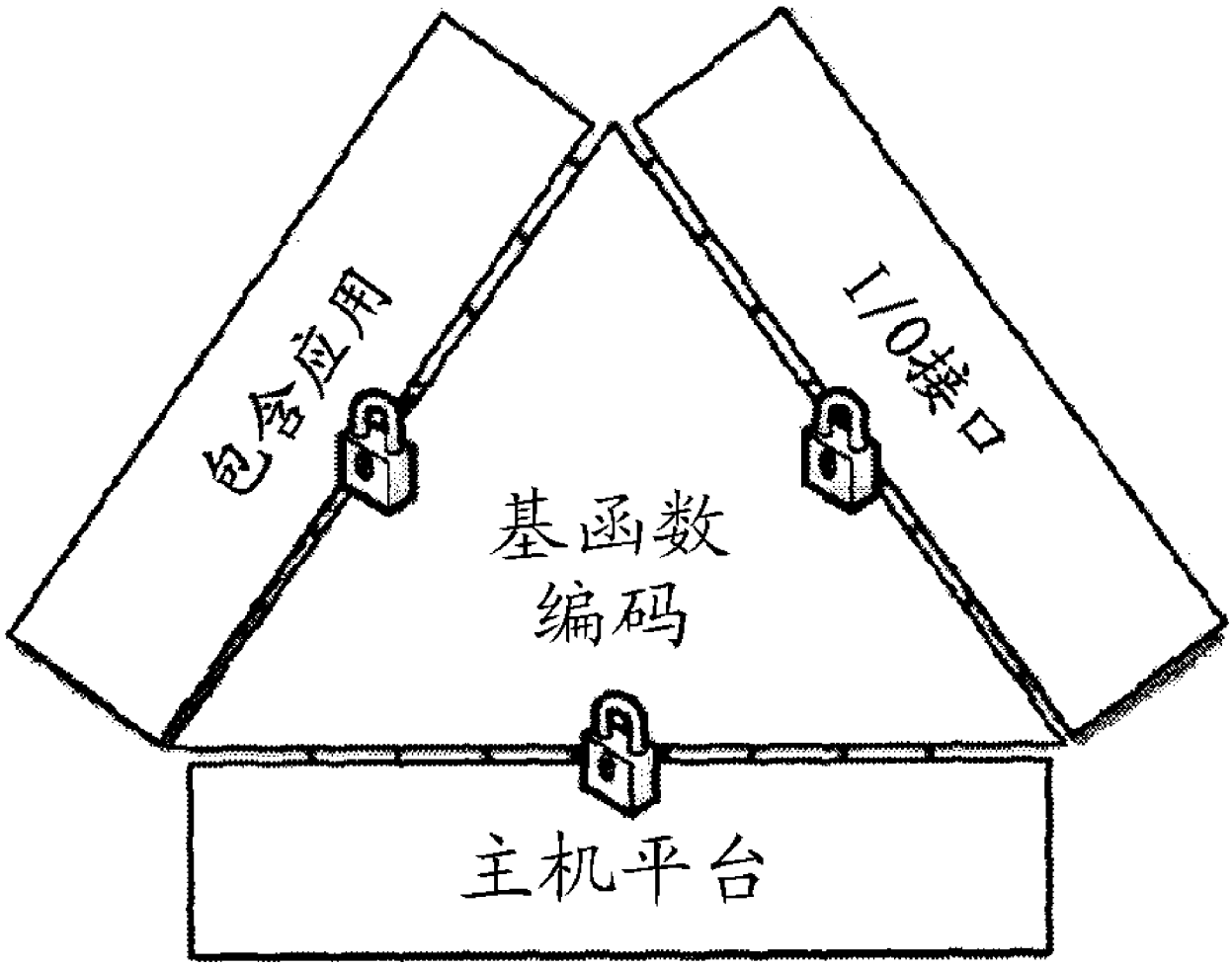


图 22

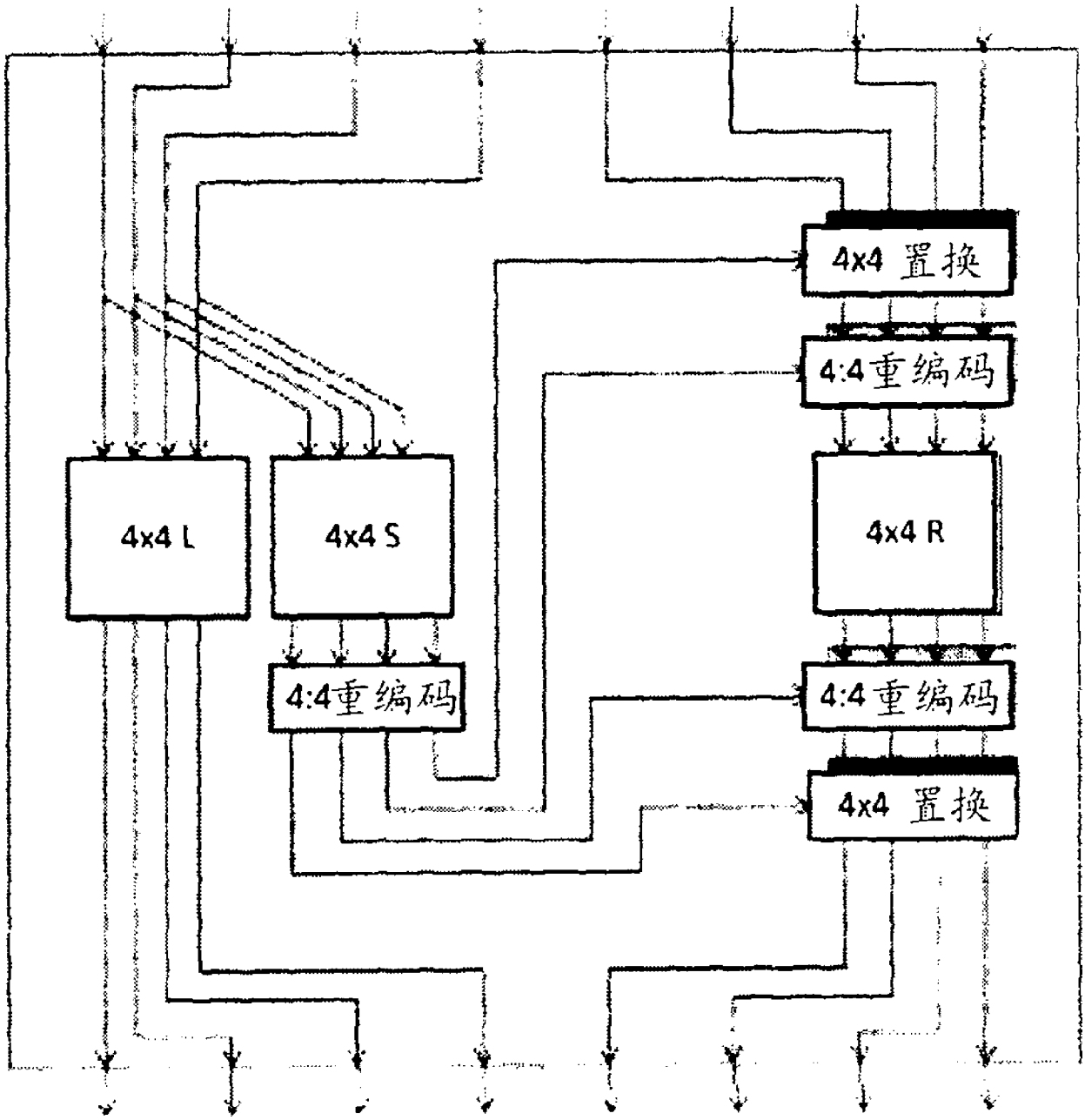


图 23

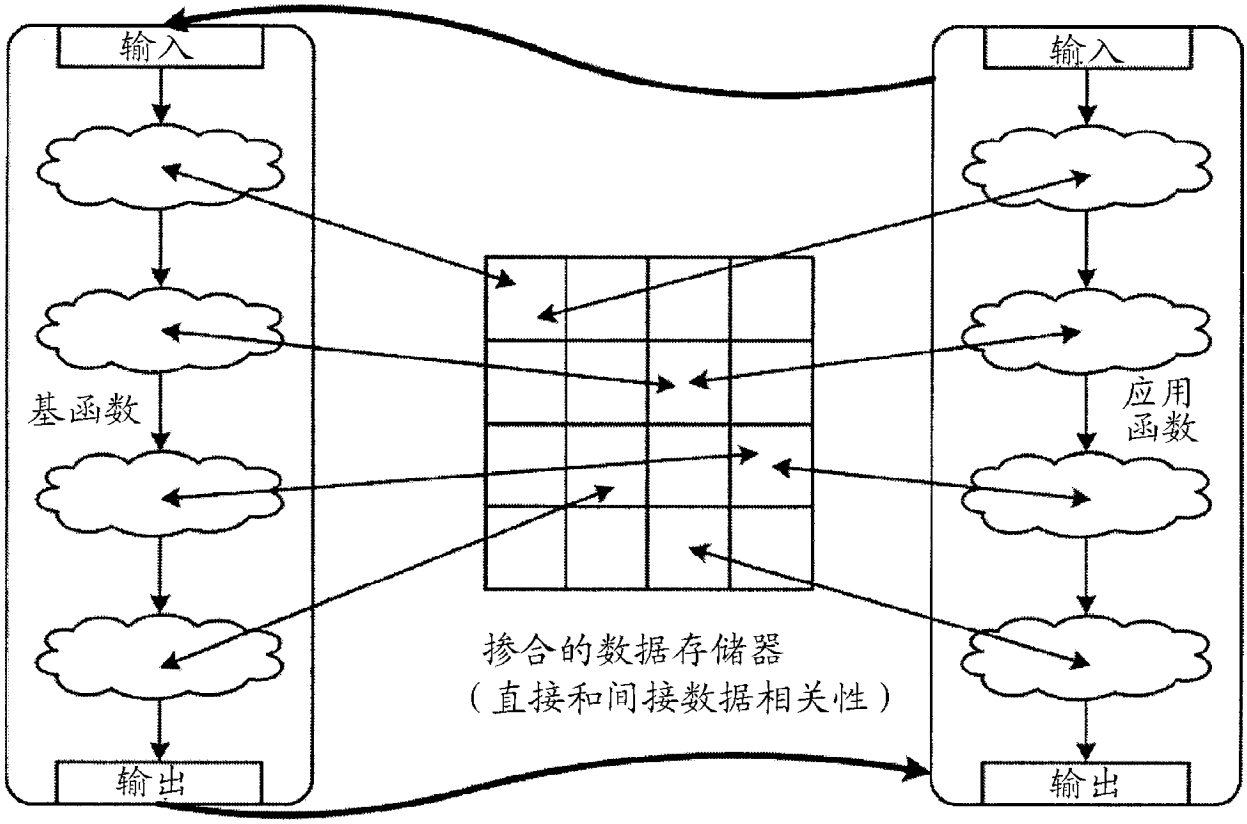


图 24

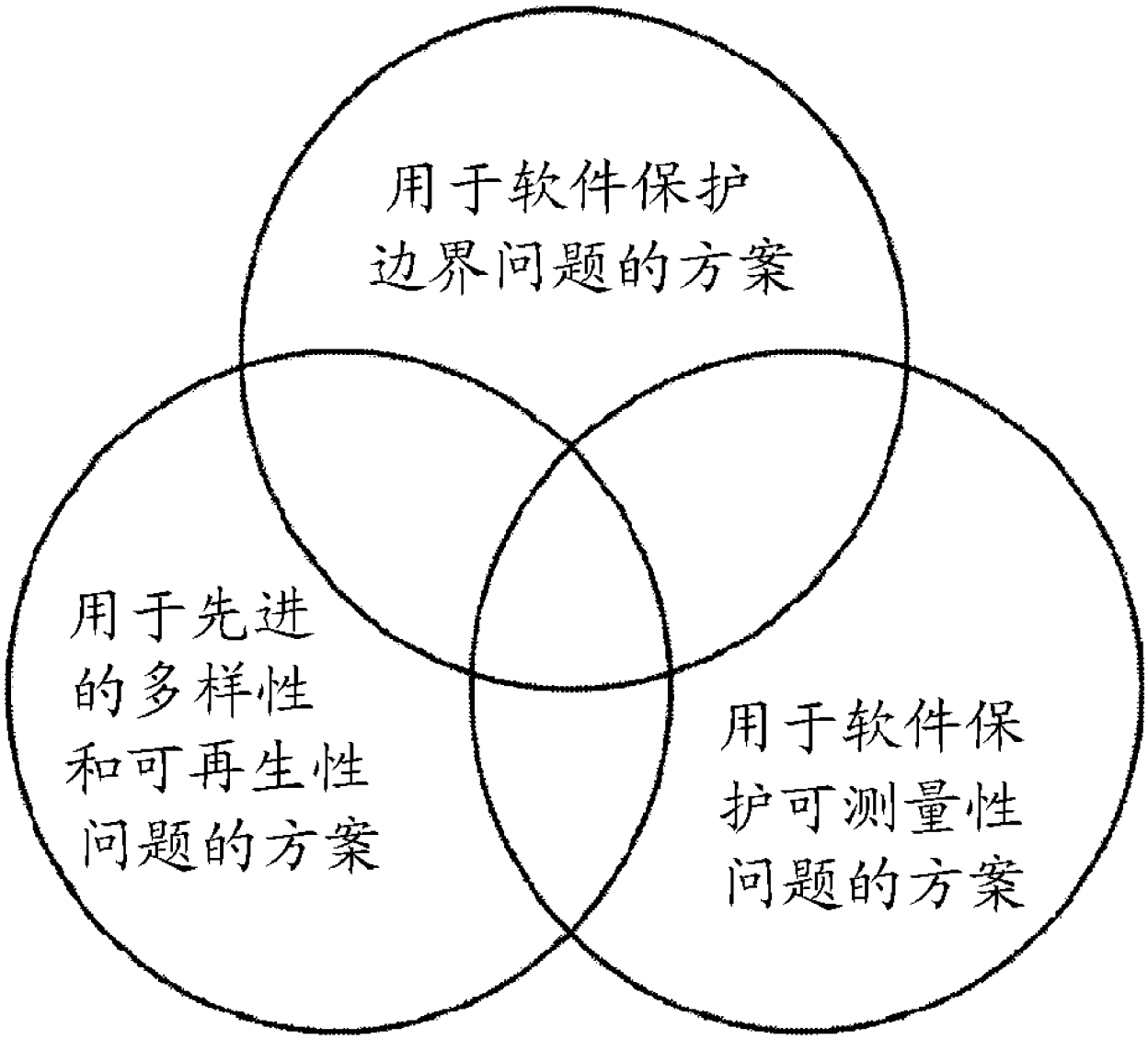


图 25

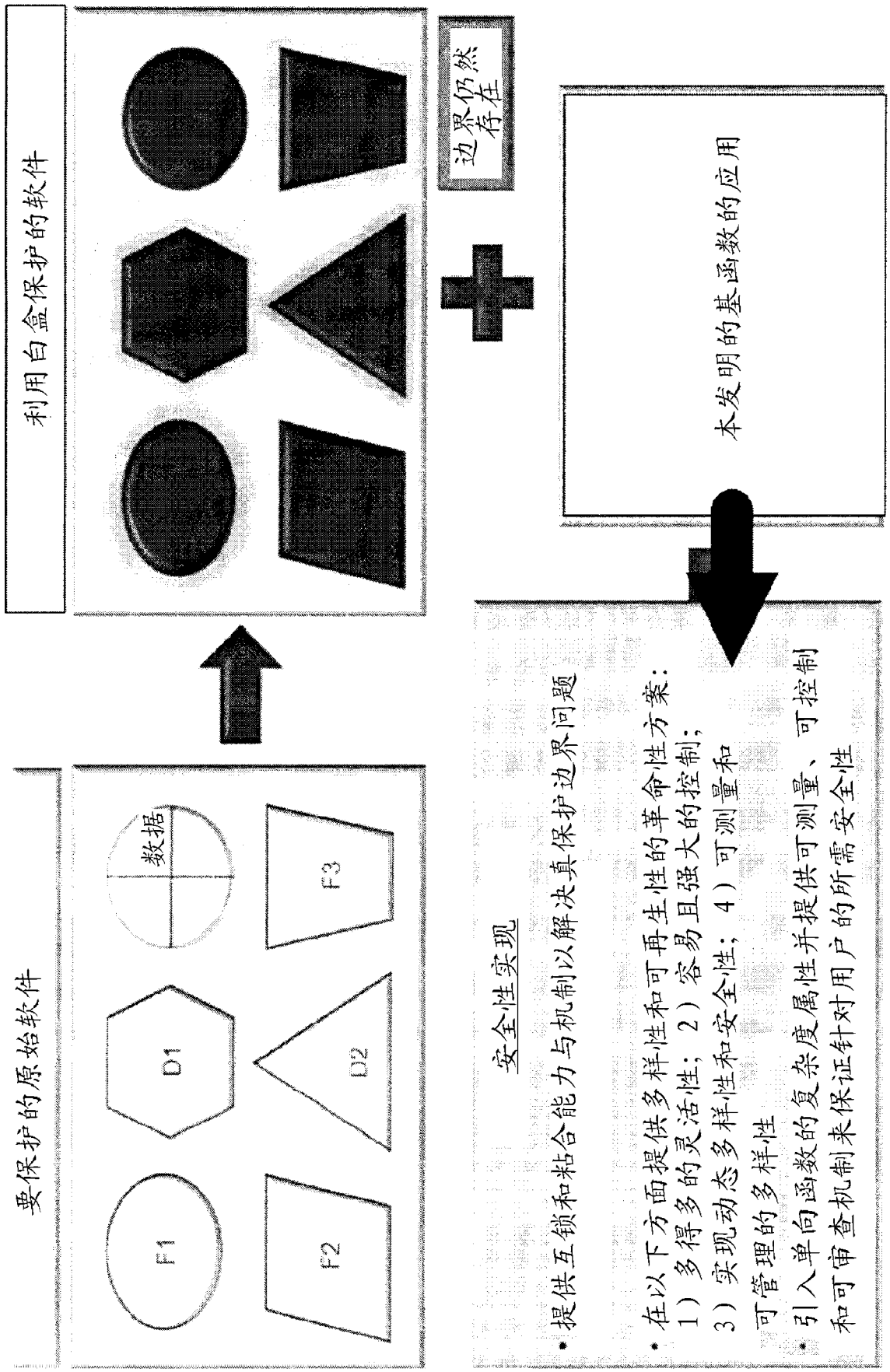

要保护的原始软件
F1
D1
数据
F2
D2
F3
利用白盒保护的软件
边界仍然存在
本发明的基函数的应用
安全性实现
• 提供互锁和粘合能力与机制以解决真保护边界问题
• 在以下方面提供多样性和可再生性的革命性方案：
1）多得多的灵活性；2）容易且强大的控制；
3）实现动态多样性和安全性；4）可测量和
可管理的多样性
• 引入单向函数的复杂度属性并提供可测量、可控制
和可审查机制来保证针对用户的所需安全性

图 26

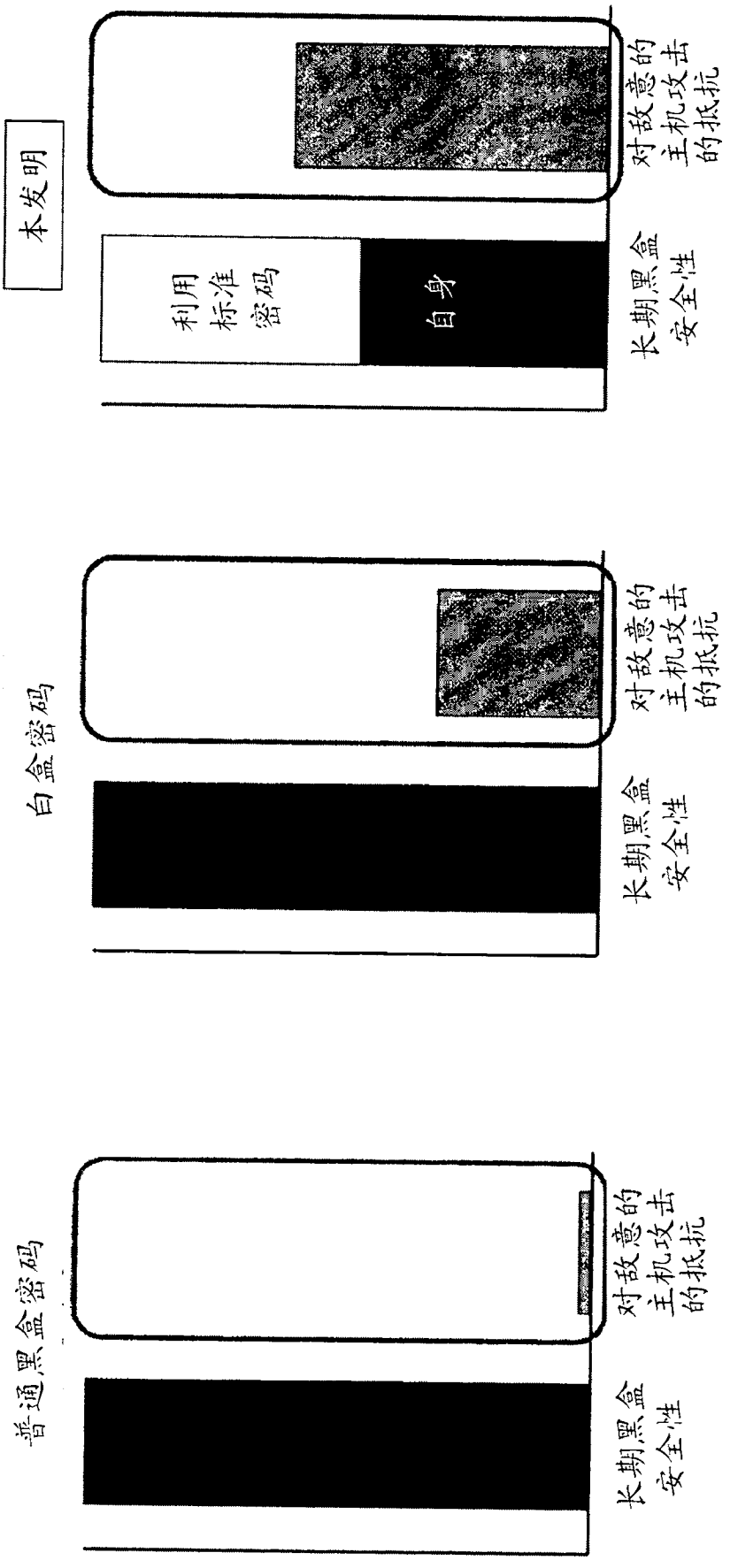

普通黑盒密码
长期黑盒安全性
对敌意的主机攻击的抵抗
白盒密码
长期黑盒安全性
对敌意的主机攻击的抵抗
本发明
利用标准密码
自身
长期黑盒安全性
对敌意的主机攻击的抵抗

图 27

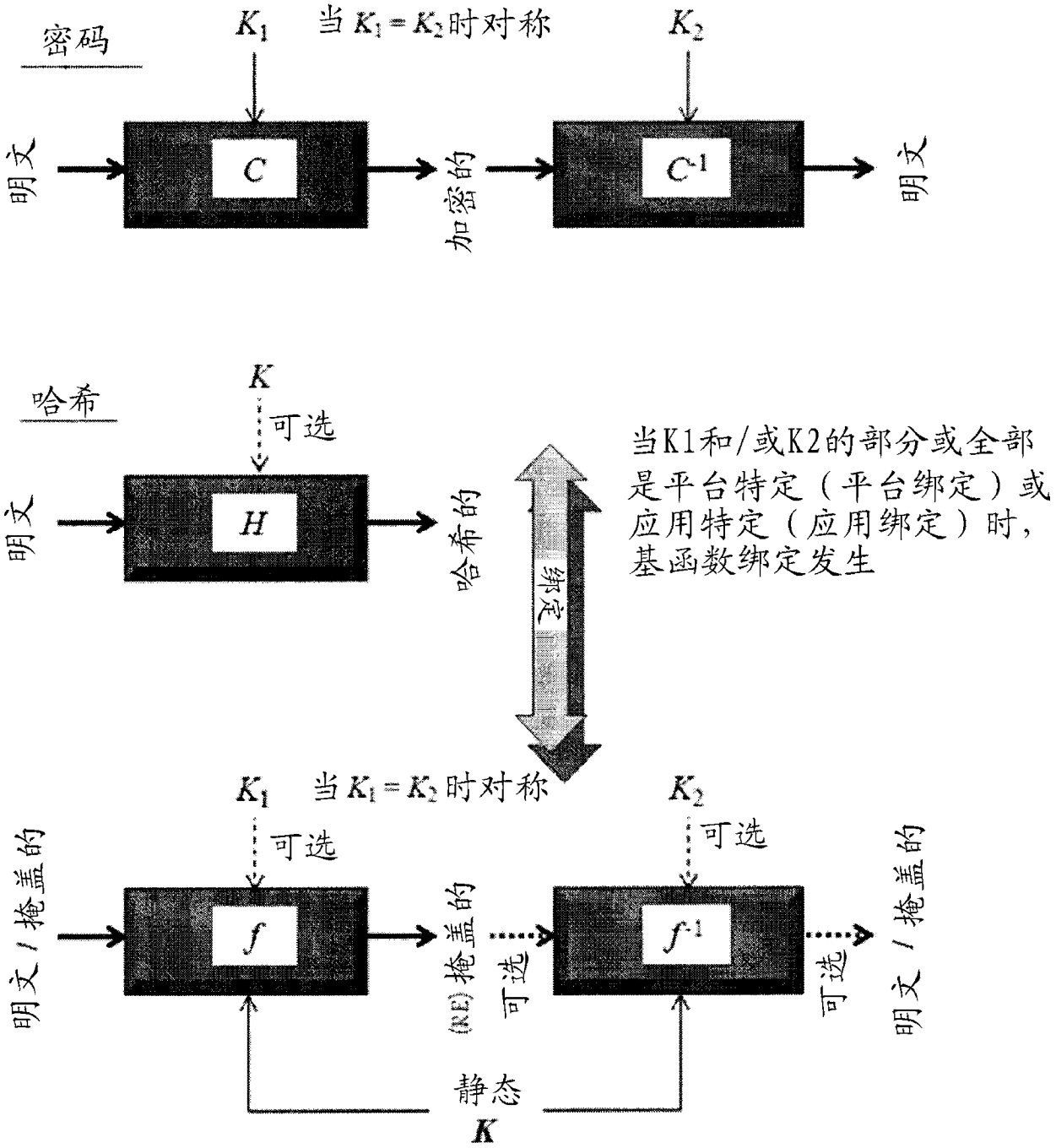

密码
K_1
当 $K_1 = K_2$ 时对称
K_2
明文
C
加密的
C^{-1}
明文
哈希
K
可选
明文
H
哈希的
绑定
当K1和/或K2的部分或全部是平台特定（平台绑定）或应用特定（应用绑定）时，基函数绑定发生
K_1
当 $K_1 = K_2$ 时对称
K_2
可选
可选
明文 / 掩盖的
f
(KE)掩盖的
可选
f^{-1}
可选
明文 / 掩盖的
静态
K

图 28

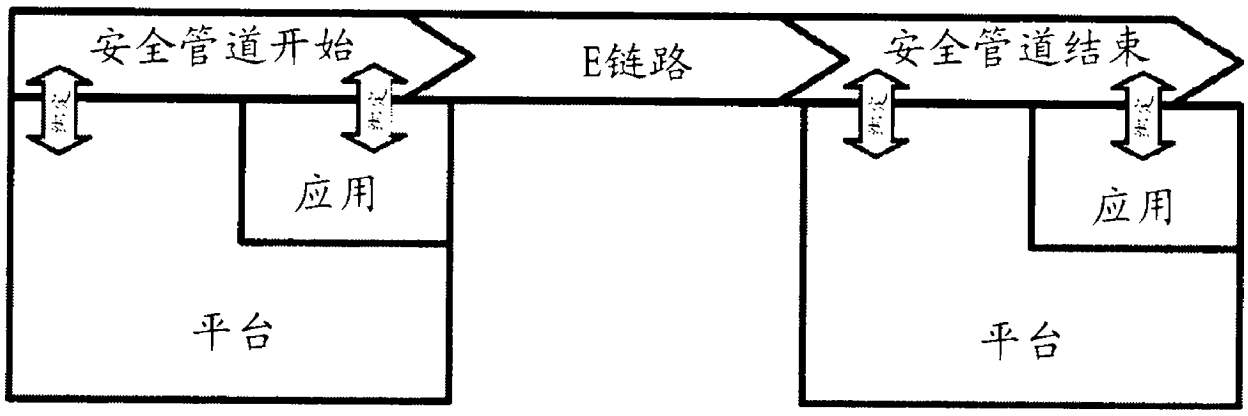

安全管道开始
E链路
安全管道结束
应用
应用
平台
平台

图 29

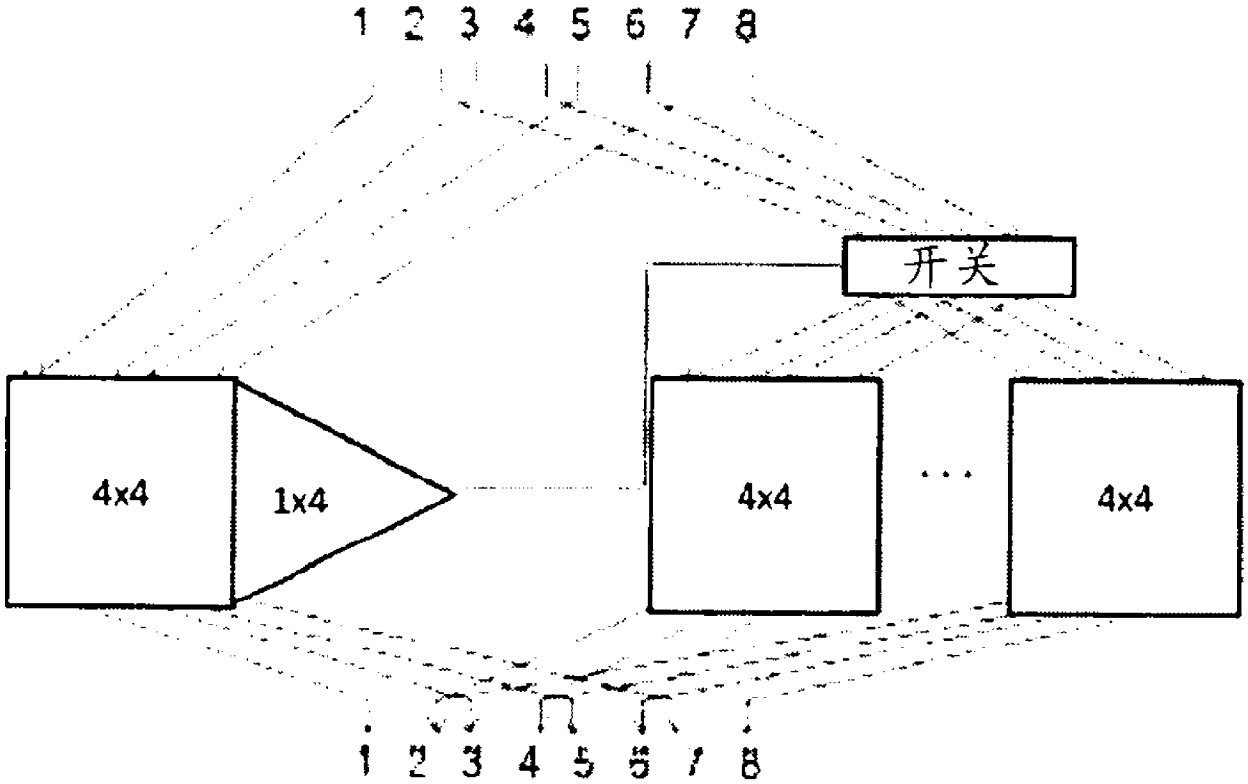

1 2 3 4 5 6 7 8
开关
4x4
1x4
4x4
...
4x4
1 2 3 4 5 6 7 8

图 30

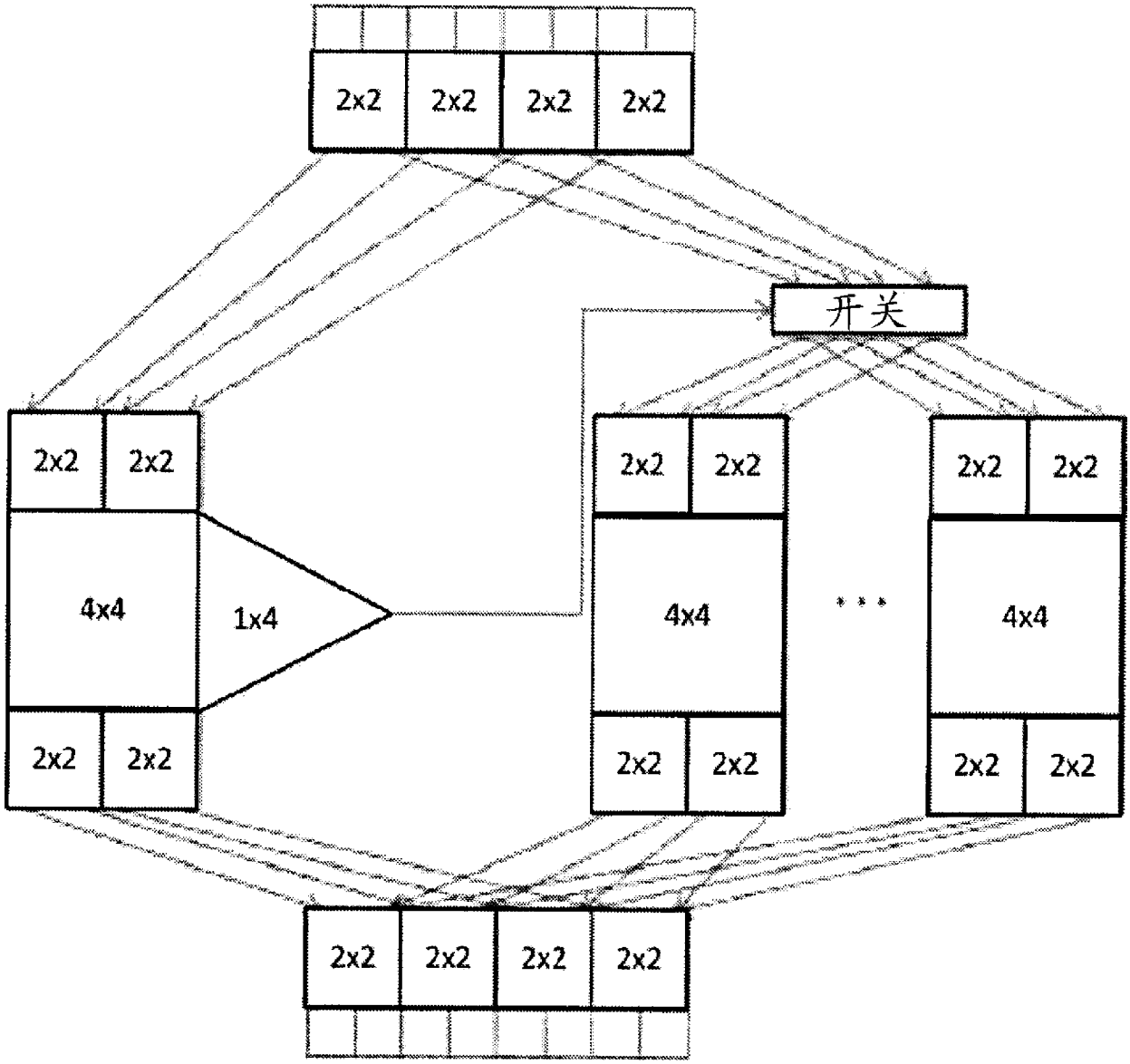

2x2
2x2
2x2
2x2
开关
2x2
2x2
4x4
1x4
2x2
2x2
2x2
2x2
4x4
2x2
2x2
2x2
2x2
4x4
2x2
2x2
2x2
2x2

图 31

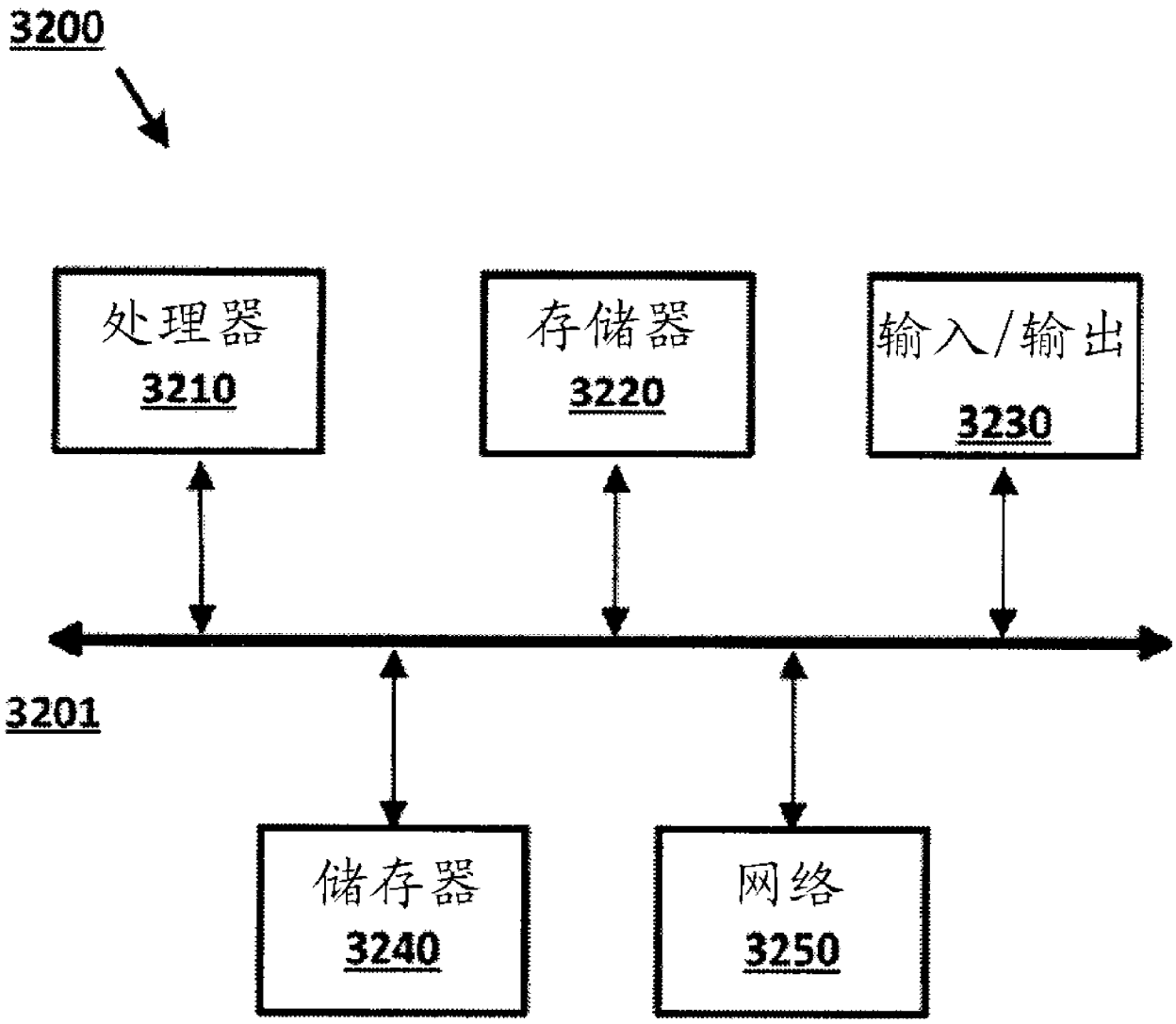


图 32