

PHANTOM:高效实现

AVIV YAISH

以色列耶路撒冷希伯来大学工程与计算机科学学院

摘要：PHANTOM[1] 是一种新的分布式交易确认协议，是一种具有可扩展性和安全性的设计，一种著名的具有可扩展性和安全权衡的区块链协议。本文介绍了一种高效的协议变体的实现，以及一种设计用于测试块DAG区块协议的仿真框架。

内容

1.实现

2.模拟框架

3.结果与结论

4.未来工作

附录

鸣谢

引文

原文：<https://github.com/AvivYaish/PHANTOM/blob/master/PHANTOM — an efficient implementation.pdf>

1.实现

这里介绍的是一个PHANTOM的贪婪实现，这意味着每个块“继承”含有过去蓝色块数量最多的父块的颜色。从中继承着色的块将被称为其着色父块。除了继承的着色之外，每个块的颜色（根据给定的着色规则）过去的所有块都不是由其着色父块着色的。

继承意味着对于每个块，在其着色父块“之后”的块的着色（和排序，给定正确的算法）是固定的。因此，仅保存改变的，diffpast为高效的着色和排序算法提供了坚实的基础，即使是所谓的“重组”，或者整编，以及由于添加了一个新块，DAG先前接受的着色定序需要进行显著更改的事件。

当你看到继承链的时候，意味着从给定的区块开始并查看其着色父块时，以及它着色父块的着色父块时，等等，一条着色链就收到了，因为该区块可以称为这条着色链的tip。因为链上的每个区块的颜色都超过了没有被其父块着色的区块数量，这个着色链会对该块的过去集中所有区块进行着色。

如[1]的算法1所示，在这个版本中，一个给定的DAG的所有tips（过去集中蓝色块最多那个区块）的蓝色集会引起整张图的着色；我们将此块称为整个DAG的着色tip。因为它的过去集的着

色是固定的，所以只剩下计算该区块的着色和定序了，而不需要重新对它过去集和反过去集中的区块进行计算。

通过观察一个其所有父块在DAG中入度都为0的“假想”块，我们知道它的着色父块是DAG的着色tip。并且添加进着色块中的区块不是由它的着色父块所着色的 – 意思正是着色父块的反过去集。把这个假想的区块叫做DAG的虚拟区块，并且请注意，它过去集的着色正是整张图的着色。

以上的表述为现在将被描述的PHANTOM的实现提供一条高效和清晰的道路。

1.1. 定义. 让我们继续对一些重要术语给出正式定义：

定义1.1. 区块 (Block)：可以保存数据的图形顶点。每一个区块都由公认的ID标识，因此作为区块的全局ID被引用，该ID是通过计算区块的哈希值所得到的。一个区块外边角所指向的区块被叫做该区块的父块。

定义1.2. BlockDAG：一个每个顶点都是区块的有向无环图。

定义1.3. 创始区块 (Genesis block)：可以认为每个BlockDAG至少包含一个区块，该区块在所有其他区块之前被添加到BlockDAG，它的全局ID为0，并且可以从所有其他区块访问，这个区块就称为创始区块。

定义1.4. 局部ID (Local ID)：给定一个DAG G，一个线性排序算法ord，以及一个被包含在G中的全局ID的区块，该块根据线性排序的局部ID是在其序列中的索引，并且将由 $\text{ord}(G, \text{block})$ 所指定。

定义1.5. DAG映射 (DAG mapping)：给定一个DAG G，一个线性排序算法ord，这个映射是一个根据G和ord通过全局ID返回局部ID的函数。这样：

$$\forall \text{block} \in G : \text{mapping}(G, \text{ord}, \text{block}) = \text{ord}(G, \text{block})$$

定义1.6. 着色 (Coloring)：给定一个DAG G，在G中选择一个k-集群(请看在 [1] 中的定义1)的过程称为对G的着色。包含在着色块中的区块就称为蓝色块，而所有其他的区块就是所谓的红色块。

定义1.7. 着色规则 (Coloring rule)：给定一个DAG G和一个着色算法color, 定义：

$$\text{coloringRule}(G, \text{color}, \text{block}) := \begin{cases} \text{True} & \text{if } \text{block} \in \text{color}(G) \\ \text{False} & \text{otherwise} \end{cases}$$

相反的，可以使用一个着色规则重新定义一个着色：

$$\text{color}(G) := \{\text{block} : \text{coloringRule}(G, \text{block}) \text{ is true}\}$$

定义1.8. 过去集 (Past)：给定一个DAG G和一个具有全局ID的区块，在G中该区块的过去集是包含所有从该区块有一条路径到达他们的区块构成的一个子图，这些区块将被 $\text{past}(G, \text{block})$ 所指定。

定义1.9. Tip：给定一个DAG G，一个G的tip是一个被包含在G中入度为0的区块。在G里给定一个全局ID的区块，用 $G!block$ 来表示，这样该块就是G的子图的唯一的tip。

定义1.10. 虚拟区块 (Virtual block)：给定一个DAG G，它的虚拟块是一个“假设”块，实际上不存在DAG中，这样该区块的过去集是整个图，意思就是它的边连接着G中的每一个tip。通过 $\text{virtual}(G)$ 来表示它。注释：

$$\text{past}(G, \text{virtual}(G)) = G$$

定义1.11. 反过去集 (Antipast)：给定一个DAG G和一个具有全局ID的区块，该区块在G中的反过去集被定义为：

$$\text{antipast}(G, \text{block}) := G \setminus \text{past}(G, \text{block})$$

定义1.12. Diffpast：给定一个DAG G，并且两个区块：block1, block2，而且 $\text{block1} \in \text{past}(G, \text{block2})$ ，block2的diffpast和block1的关系定义如下：

$$\text{diffpast}(G, \text{block2}, \text{block1}) := \text{past}(G, \text{block2}) \setminus \text{past}(G, \text{block1})$$

定义1.13. 蓝Diffpast (Blue diffpast)：给定一个DAG G，一个着色算法color，以及两个区块block1 和block2，并且 $\text{block1} \in \text{past}(G, \text{block2})$ ，block1的蓝diffpast根据G和color被定义如下：

$$\text{blueDiffpast}(G, \text{color}, \text{block2}, \text{block1}) := \text{diffpast}(G, \text{block2}, \text{block1}) \cap \text{color}(G)$$

通常，block1将是block2的着色父块，并且它的着色将是由block2的着色链引发。在这种情况下，将使用以下速记：

$$\text{blueDiffpast}(G, \text{block2})$$

定义1.14. 红Diffpast (Red diffpast)：同样的，我们定义：

$$\text{redDiffpast}(G, \text{color}, \text{block2}, \text{block1}) := \text{diffpast}(G, \text{block2}, \text{block1})$$

$$\setminus \text{blueDiffpast}(G, \text{color}, \text{block2}, \text{block1})$$

定义1.15. 着色链 (Coloring chain)：给定一个DAG G和一个区块，着色链是由区块按照一定序列组成的链，区块n是区块n-1的着色父块，例如：

$$\forall i \in [n - 1] : \text{coloringParent}(\text{block}_{i+1}) = \text{block}_i$$

为简便起见，给定两个区块b, b' 并且b是b'的祖先，链从b'延伸到b的部分也可以写成：b' → b，而以b'开头的整个着色链为：coloringChain(b')

定义1.16. 主着色链 (Main coloring chain)：给定一个DAG G，它的主着色链是由所有蓝色过去集最大的tip组成的，用它表示：coloringChain(G)

定义1.17. 链差 (Chain difference)：给定两条链 $b' \rightarrow b, c' \rightarrow c$ ，该两条链之间的差异定义如下：

$$(b' \rightarrow b) - (c' \rightarrow c) := ((b' \rightarrow b) \cup (c' \rightarrow c)) \setminus ((b' \rightarrow b) \cap (c' \rightarrow c))$$

备注：在大多数情况下，G是整个DAG。因此，在这种情况下，将省略G简写参数。

1.2. 算法

定义1.2.1 贪婪的PHANTOM数据结构：贪婪的PHANTOM数据结构为具有全局ID的每个区块保存以下属性：

- 外向的边缘
- 着色父块，它是区块继承而来的着色的区块
- 区块的高度
- 在G中的着色块具有蓝色着色块的总数
- 区块的蓝diffpast和红diffpast，并且每一个都依照区块在G中的序号来形成的区块在diffpast中的索引号保存为映射。
- 在G中区块自己的序号

此外，蓝色和红色的diffpast保存为G的虚拟块，还有一点区别：虚拟块的diffpast尽管保持最新（见算法5），每添加一个区块，并不是依照虚拟的着色链去着色和排序的，因为它与添加新块完全无关，只有在接收到有关虚拟块diffpast中某个块的顺序的查询时才是有必要的。因此，如果需要，当接收到这样的查询时，将计算虚拟的diffPast的颜色和顺序。

算法1-7详细描述了数据结构的处理。

备注：在内存方面保存所有块的diffpast的颜色和顺序是非常低效的。但是，在主着色链上的一个块被接受之后，比如说区块，那么它的所有过去也都被接受了，考虑到协议的贪婪性质以及它对DAG中“足够深”的块着色的收敛性，意味着为所有块保留所有这些数据是不必要的。因此，可以安全地删除过去所有块的蓝色和红色diffpast（着色链上的块除外），因为只要块仍然是主着色链的一部分，它就不会被使用。

如果再次需要修剪的数据，可以使用最初计算数据的相同算法重新计算，从最近的主着色链块开始重新填充所需数据并向外扩散。因此，在设置了接受阈值之后，在主着色链每延长一个块之后，可以为块（红色和蓝色）diffpast中的所有块修剪DAG的数据结构。这种迭代修剪确保数据结构的大小保持在最小值。

1.2.2 着色算法：[1]的算法1可以根据diffpass递归地重新定义，并将其推广为使用任意着色规则而不是基于反锥体的规则，如算法9-11所示。

Algorithm 1: CREATE-BLOCK(*parents*, *data*)

Input : *parents* - the parents of the block,
 data - the data to be contained in the block

Output: *block* - a block with the given properties

- 1 *block* $\leftarrow \emptyset$
 - 2 *parents(block)* $\leftarrow \text{parents}$
 - 3 *data(block)* $\leftarrow \text{data}$
 - 4 return *block*
-

Algorithm 2: CREATE-DAG(*coloringRule*)

Input : *coloringRule* - the coloring rule for the DAG

Output: *G* - a PHANTOM block-DAG

- 1 *V* $\leftarrow \emptyset$
 - 2 *E* $\leftarrow \emptyset$
 - 3 *G* $\leftarrow (V, E)$
 - 4 *rule(G)* $\leftarrow \text{coloringRule}$
 - 5 *genesis(G)* $\leftarrow \emptyset$
 - 6 *virtual(G)* $\leftarrow \text{CREATE-BLOCK}(\emptyset, \emptyset)$
 - 7 *coloringChain(G)* $\leftarrow \emptyset$
 - 8 return *G*
-

Algorithm 3: GET-COLORING-PARENT(*G*, *block*)

Input : *G* - a PHANTOM block-DAG,
 block - a block in *G*

Output: *coloringParent* - the parent block which maximizes the number of
blue blocks in its past

- 1 return $\arg \max\{\text{blueNumber}(\text{parent}) : \text{parent} \in \text{parents}(\text{block})\}$
 - /* Break ties according to lower global ID */
-

Algorithm 4: ADD-BLOCK(*G*, *block*)

Input : *G* = (*V*, *E*) - a PHANTOM block-DAG,
 block - a block

Output: *G* - the graph after *block* was added to it

- 1 *G* $\leftarrow (V \cup \text{block}, E \cup \{(\text{block}, \text{parent}) : \text{parent} \in \text{parents}(\text{block})\})$
 - 2 *coloringParent(block)* $\leftarrow \text{GET-COLORING-PARENT}(\text{block})$
 - 3 *height(block)* $\leftarrow \text{height}(\arg \max\{\text{height}(\text{parent}) : \text{parent} \in \text{parents}(\text{block})\}) + 1$
 - 4 *color(diffpast(block))* $\leftarrow \text{CALC-DIFFPAST-COLOR}(\text{block})$
 - 5 *blueNumber(block)* $\leftarrow \text{blueNumber}(\text{coloringParent}(\text{block})) + |\text{blue}(\text{diffpast}(\text{block}))|$
 - 6 *order(diffpast(block))* $\leftarrow \text{CALC-DIFFPAST-ORDER}(\text{block})$
 - 7 *selfIndex(block)* $\leftarrow \max(\text{order}(\text{diffpast}(\text{block}))) + 1$
 - 8 *G* $\leftarrow \text{UPDATE-MAX-CHAIN}(\text{block})$
 - 9 return *G*
-

要从这里介绍的算法11接收[1]的算法1，只需使用算法2和以下规则创建一个新的PHANTOM DAG区块作为输入：

$$(1) \quad rule_k(G, b) = \begin{cases} True & |anticone(G, b) \cap \text{GET-BLUE}(G)| \leq k \\ False & otherwise \end{cases}$$

然后，对于指定的k值，在使用算法4构造的图G上调用算法11将产生与在G上调用[1]中算法1相同的输出。

为了高效地将块添加到PHANTOM DAG中，需要高效地实现其着色规则。但是，很难根据任意着色来计算任意块的蓝色反锥体的大小，这意味着

$$|anticone(G, b) \cap \text{GET-BLUE}(G)|$$

对于区块b，即使它“合理地”靠近G的叶。

在这里介绍的实现中，没有使用规则1来代替高效的方法和数据结构来检索上述内容。

$$(2) \quad rule_k(G, b) = \begin{cases} True & coloringChain(b) \cap \text{CALC-K-CHAIN}(G, k) \neq \emptyset \\ False & otherwise \end{cases}$$

与规则1一样，规则2“惩罚”扣留的块，而这里的度量是G主链上的块引用的蓝色块的数量，而不是tip为b的着色链上的块。结合主着色链的贪婪选择规则，这鼓励矿工将当前已知的所有叶作为新开采块的父级，并尽快发布。

使用算法9可以有效地计算此规则：给定k链，其中包含的最低高度块的高度为h，给定块的着色最多需要O(height(b) - h)。

可以考虑更有效的可计算规则，例如：

$$(3) \quad rule_k(G, b) = \begin{cases} True & \text{CALC-K-CHAIN}(coloringChain(b), k) \cap \\ & \text{CALC-K-CHAIN}(G, k) \neq \emptyset \\ False & otherwise \end{cases}$$

平均O(k)。

注意，当在给定的块上运行时，算法10执行以下操作一次：

- 计算antipast(coloringParent(block))：区块的反过去集是通过取虚拟块的反过去集得到的，是在只在虚拟块和块的着色链上行走的同时，添加/删除从虚拟块到块的路径上的块的差异而得到的，如7所示。反过去集的计算如下：首先，找到了coloringParent(block)着色链和G主着色链中最高的块；平均值为：O(|coloringChain(block) - coloringChain(G)| ∩ coloringChain(block)|)

然后是路径上的每个块的diffpast：coloringParent(virtual(G)) → intersection

被延迟（见附录4）地添加到diffpast和路径上每个块的diffpast中：

$$\text{coloringParent(block)} \rightarrow \text{intersection}$$

从反过去集中延迟删除。总之，我们得到的平均运行时复杂度为：

$$O(|\text{coloringChain(block)} - \text{coloringChain}(G)|)$$

- 区块k链的计算：除了创世块之外，每个块在其着色父块（着色父块本身）的着色中至少添加一个蓝色块；因此，k链的长度至多为k。因此，计算k链（通常在实际代码中预先计算）平均需要O(k)。

以及对于每个块的以下操作，当前块是作为diffpast中的BFS的一部分的。

- 安全检查 $\text{antipast}(\text{coloringParent(block)})$ ：检查所用的时间与计算反过去集的时间相同，因为Lazyset只是根据添加/删除（见附录4）的顺序遍历添加到它的每个集合。
- 区块的k链和当前区块的着色规则的激活：用 $O(|rule|)$ 来表示

假设 $\text{diffpast}(\text{block}) = (V, E)$ ，因此，总的来说，算法10取平均值

$$O(|E| + |V| \cdot k \cdot \underbrace{|\text{coloringChain}(\text{block}) - \text{coloringChain}(G)|}_{*} \cdot |rule|)$$

请注意，当不涉及违规行为时，由于上述诚实矿工的激励，*应该相对较小。

备注：当PHANTOM DAG的区块数据结构保存包括虚拟块的每个块的蓝色diffpast时候，根据贪婪选择规则，整个图的着色是通过沿着主着色链将每个块的diffpast的着色链接在一起给出的，从虚拟块开始，结束于创世块，如算法11所计算。

在实际的代码中，使用对主着色链的diffpast着色的引用来不断更新链表。因此，尽管表和索引的数据整理是高效的（平均取O(链差)）但包含检查平均取O(主着色链的长度)。

整个图的着色和着色tip的diffpast只对有关区块的顺序的查询有作用，并且根本不参与新块的添加；因此，这是可以接受的。但是，如果需要加速，可以保存所有已被接受的主链区块的过去的着色集，因为该集在未来不太可能改变，类似于备注1.2.1中提到的数据结构调整。

1.2.3 排序算法：与着色算法一样，所提出的排序算法允许每个块继承其着色父块的顺序，并将其扩展到它的diffpast。通过首先查看着色父块，该算法得出了一个对过去的块不可更改的顺序，从而可以与每个块的diffpast着色一起保存。

当使用算法4添加块时，排序算法在执行算法10之后运行，因此区块的红色和蓝色diffpast就相应的已经计算好了。算法14简单地遍历diffpast，根据以下递归规则为每个块分配一个索引：给定一个区块 b 和着色diffpast，coloringParent(b)具有最低的可定序的索引，下面的索引是给 $\text{color}(\text{diffpast}) \cap \text{parents}(b)$ 的，较低的索引是给全局ID较低的块的，剩下的是给 $\text{parents}(b) \setminus \text{color}(\text{diffpast})$ 。同样，对于全局ID较低的块，使用较低的索引，停止条件是到达一个已经排序的块。

假设该区块的diffpast是有序的，我们知道coloringParents(block)在之前已经定好序了，于是我们也知道coloringParents(block)自己在图中序列里的索引，这个索引将根据区块的顺序（当然是coloringParents(block)本身）给出的第一个diffpast的块，因为coloringParents(block)的所有父块在添加到图中之前都已被定序好了。

此递归规则对每个块执行BFS遍历，执行

$$O(|\text{parents}(block)| \log |\text{parents}(block)|)$$

每个区块的操作。假设 $\text{diffpast}(block) = (V, E)$ ；总的来说，该算法的平均运行时间复杂度为

$$O(|E| + |V| \cdot |E| \log |E|)$$

注意，排序只需要确保知道相同块的所有矿工以相同的顺序遍历它们，从而使它们的顺序同步。但是，如果区块数据结构使父集保持有序，并且全局ID是在整个数据结构（包括父集）上计算的，那么算法13即使不使用排序，每次也会产生相同的顺序，避免了排序的需要，并且给算法14一个总的平均复杂度为 $O(|V| + |E|)$ 。

备注：如备注1.2.2所述，整个DAG区块的顺序实际上保存在一个链图中，对块的本地ID进行查询，其效率、问题和解决方案与对图主着色块的颜色进行查询相同。

1.2.4 时间复杂度：算法5：假设添加了一个新的块 b ，如果它不是图中的新的着色链的tip，那么运行时间为 $O(|\text{parents}(b)|)$ ，在第2行。

如果它是新的tip，则更新图的着色链。表示如下：

$$b \rightarrow b' := \text{coloringChain}(b) - \text{coloringChain}(b')$$

如果前一个着色块是 b' ，则平均值为

$$O\left(\sum_{c \in b \rightarrow b'} |\text{diffpast}(c)|\right)$$

由于antipast(b)的计算。注意假设没有犯规，这应该是小的，因此是高效的。

请注意， $\text{parents}(b)$ 包含在 $\text{diffpast}(b)$ 中。

算法4：假设在G中添加了一个新的块b，则b'是G的前一个着色tip，假设 $\text{diffpast}(b) = (V, E)$ ，根据前面第1-3行计算取 $O(|\text{parents}(b)|)$ ，第4-7行取

$$O(|E| + |V| \cdot k \cdot |b \rightarrow b'| \cdot |\text{rule}|)$$

如果b不是图的新着色tip，那么第8行显然取 $O(|\text{parents}(b)|)$ ，并且

$$O\left(\sum_{c \in b \rightarrow b'} |\text{diffpast}(c)|\right)$$

如果是这样。于是在最坏的情况下，运行时间是：

$$O\left(\sum_{c \in b \rightarrow b'} |\text{diffpast}(c)| + |V| \cdot k \cdot |b \rightarrow b'| \cdot |\text{rule}|\right)$$

2 . 模拟框架

2.1 模拟框架：为了测试PHANTOM的实现，构建了一个模拟框架，模拟矿工和网络行为，并允许对所有相关参数进行细粒度控制。

当模拟开始时，根据给定的参数，模拟网络中填充了矿工。此外，还可以模拟在动态矿工加入和离开网络。

模拟的矿工会收到一个空的DAG块作为参数，假设它实现了一个DAG抽象类，该类包含用于DAG矿工交互的最小接口。因此，根据这个接口实现的任何DAG块也可以被模拟。

该模拟将一个矿工分配给一个给定速率的泊松过程中的一个区块，在泊松过程中，矿工的选择是根据整个网络的散列速率分布来完成的。由于模拟不包括区块的交易，因此矿工只需生成一个带有随机数据的块，并根据所使用的DAG选择父块，然后继续将该块传输给所有对等方。网络为每个矿工分配随机对等点，并根据给定参数分配随机延迟给所有对等链接。

对等方接收该块并将其添加到自己的DAG（如果认为有效），根据创建时指定的行为，可以将该块广播给所有对等方（如[1]中所示），也可以不做任何操作。

如果矿工听到由于并非所有块的父块都在矿工的DAG中而无法添加的有效块，则会将其添加到要添加的队列中。现在，根据矿工参数中指定的行为，它可以向所有邻居请求失踪的父块，或者只是被动地等待，直到其中一个邻居无端地广播它。

该框架允许模拟各种攻击。例如，给出了一个自私矿工的实现方法，在这种方法中，矿工在其DAG的规则指示之前，一直拒绝发布开采块。此外，恶意矿工的网络行为也可以被控制，允许他们对对等方的网络延迟为零，或者与网络上的所有矿工有传出的链接。

2.2 模拟设置

2.2.1 参数：所有模拟均使用以下参数运行：

- 区块创建速率：1块/分钟。
- 100名诚实的矿工，每个都有8名对等邻居。
- 网络拓扑和散列率随机分配给每个模拟，但保持恶意矿工的散列率不变。
- 诚实的矿工会广播他们收到的每一个区块，但不会自动获取他们没有的区块。
- 恶意矿工具有零网络延迟和到网络上所有矿工的传出链接。
- 模拟长度设置为允许恶意矿工至少3次成功攻击。

2.2.2 攻击描述：有一个恶意矿工自私地挖矿，试图通过在私有位置启动一个与被攻击块平行的“竞争”链来执行双花攻击，试图运行诚实链，希望改变图的拓扑顺序，使第一个恶意块先于被攻击块。如果诚实的矿工接受了被攻击的区块，并且恶意矿工挖掘了足够的区块，使得第一个恶意区块在完全公开竞争链后被接受，那么竞争链将广播给网络上的所有矿工。

对于每一个新的恶意块，恶意矿工选择上一个恶意块和最靠近tip的诚实块，使其过去蓝色集小于上一块的，因此我们保证上一个块将是新恶意块的着色父块。有关更多详细信息，请参阅算法16。表示为 $\text{antipast}(\text{prev}) = (V, E)$ ，注意算法的运行时间为 $O(|V| + |E|)$ 。

根据给定的父块选择规则，如果恶意块是DAG的着色端，则着色链将通过第一个恶意块，该恶意块与被攻击的块平行。根据排序算法，这意味着恶意块的本地ID将低于被攻击的块，因此这两个块中的任何冲突的交易对都将根据恶意块来解决。只有在诚实的矿工接受诚实的区块并且恶意的矿工接受恶意的区块之后，才能发布恶意链，从而成功地执行双重花费攻击。

如果恶意矿工认为诚实链通过其恶意链的散列率超过其自身散列率乘以块确认深度，则恶意矿工将中止并重新启动攻击。

3 . 结果与结论

测试k、块确认深度和恶意散列率（与整个网络相关）的各种值，其中至少模拟每个参数组合，直到达到5%的误差范围和不确定性。

如图1-3所示，正确的k参数化和接受深度对于确保协议的安全性很重要，如假设的那样，随着k值的降低和接受深度的增加，协议的安全性也会增加。

注意，尽管 $k > 0$ 需要较大的确认深度（与区块链相比），但PHANTOM协议允许通过将块创建速率设置为更高来快速满足该深度。有很多建议（如[2]中所述）使用协议中的DAG性质来允许交易的高吞吐量，同时保持较低的块大小，从而允许矿工之间的快速传输。

4 . 未来工作

交易模拟：将交易合并到模拟中，模拟矿工对交易的选择，以及测量算法的交易和块吞吐量，对于作为加密货币的一部分的协议的实际实现来说，可能会引起相当大的兴趣。例如，对于吞吐量和交易确认时间，规则2和规则3是很有趣的。

设计更好的攻击：这里提出的攻击并没有被证明是最佳的双花攻击。我们可以考虑各种可能更好的建议，例如更改父块的选择方案（例如，与其只选择较少的“蓝色”父块，还可以选择带有不通过被攻击块的着色链的父块），甚至公共攻击。

互联网模拟：这里介绍的模拟框架并没有使用“真实”的互联网网络进行矿工之间的通信，而是模拟延迟本身。但是，Miner类将网络功能用作黑盒，允许轻松实现实际使用Internet的网络模块，允许在实际情况下进行模拟。

附录

暴力攻击PHANTOM：除了这里介绍的PHANTOM高效的贪婪实现之外，代码中还实现了PHANTOM的“原始”版本，它使用规则1以及图的着色，从而在图中实现尽可能多的蓝色块。但是，由于该算法是NP中的算法，因此是使用蛮力实现的。

LazySet：作为高效计算区块的反过去集的一部分，开发了Lazyset数据结构。与常规集合一样，用户可以检查集合中是否包含某个项目，删除和添加项目，并执行集合联合、差异、交叉等操作。

但是，当一个规则集 s 与另一个集 s' 执行联合/差异时，比如 $|s'| = n$ 的复杂度为 $O(n)$ ，则LazySet的复杂度为 $O(1)$ ，权衡控制检查运行时的复杂度，这最多需要复杂度 $O(\text{参与联合/差异的集合})$ 。

此外，请注意，Lazyset不会复制它所组成的任何集合中包含的项目，只需保留对每个集合的引用，从而节省内存。

更多信息可以在[这里](#)找到。

鸣谢

作者要感谢Yonatan Sombolinsky和Aviv Zohar对PHANTOM协议的帮助和指导，以及和Tamir Segal和Chaim Hoch的聊天对公司的帮助和指导。

图 1 使用规则 2 时，攻击成功率是关于 k 的函数：

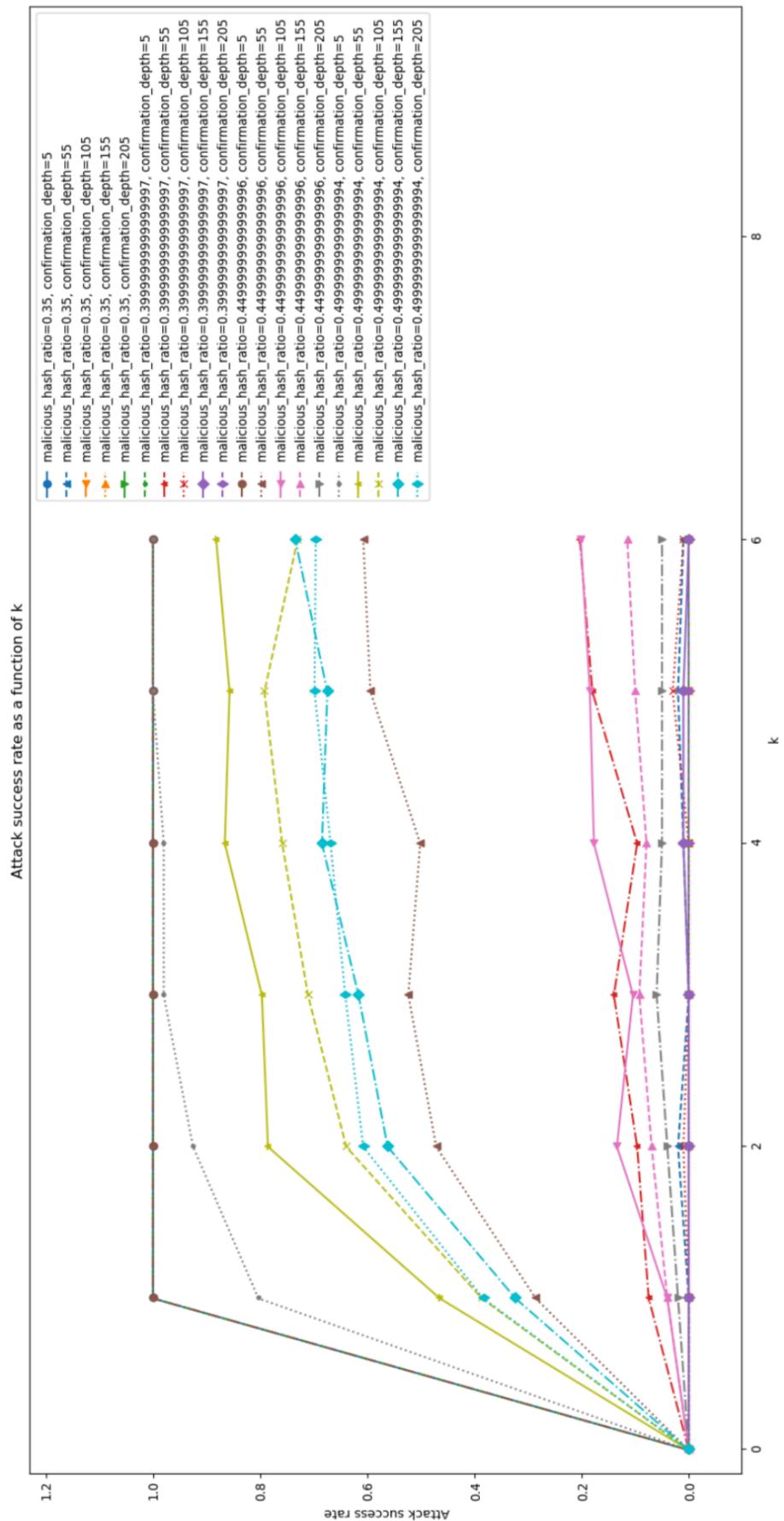


图 2 使用规则 2 时，攻击成功率是关于确认深度的函数：

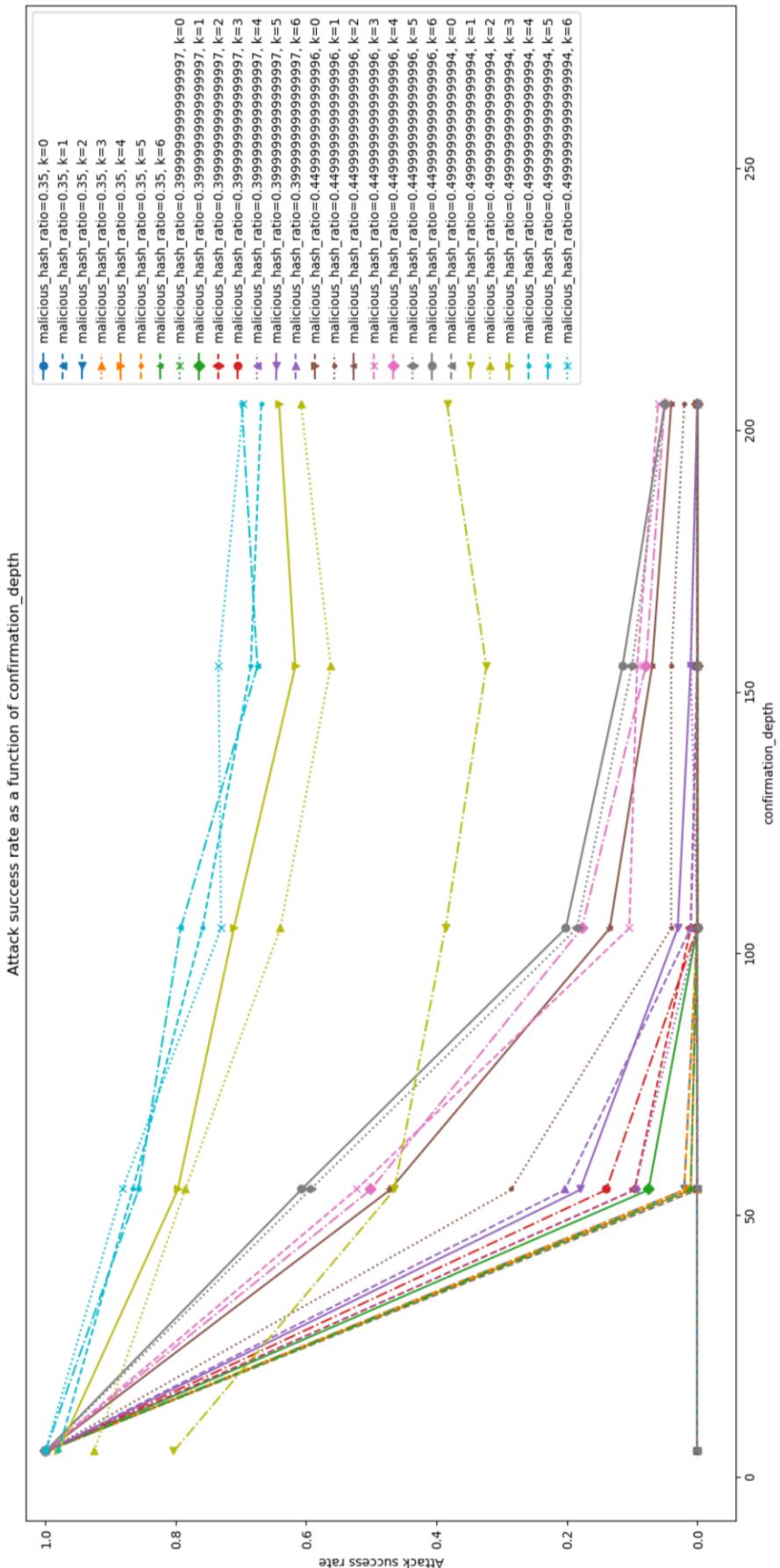
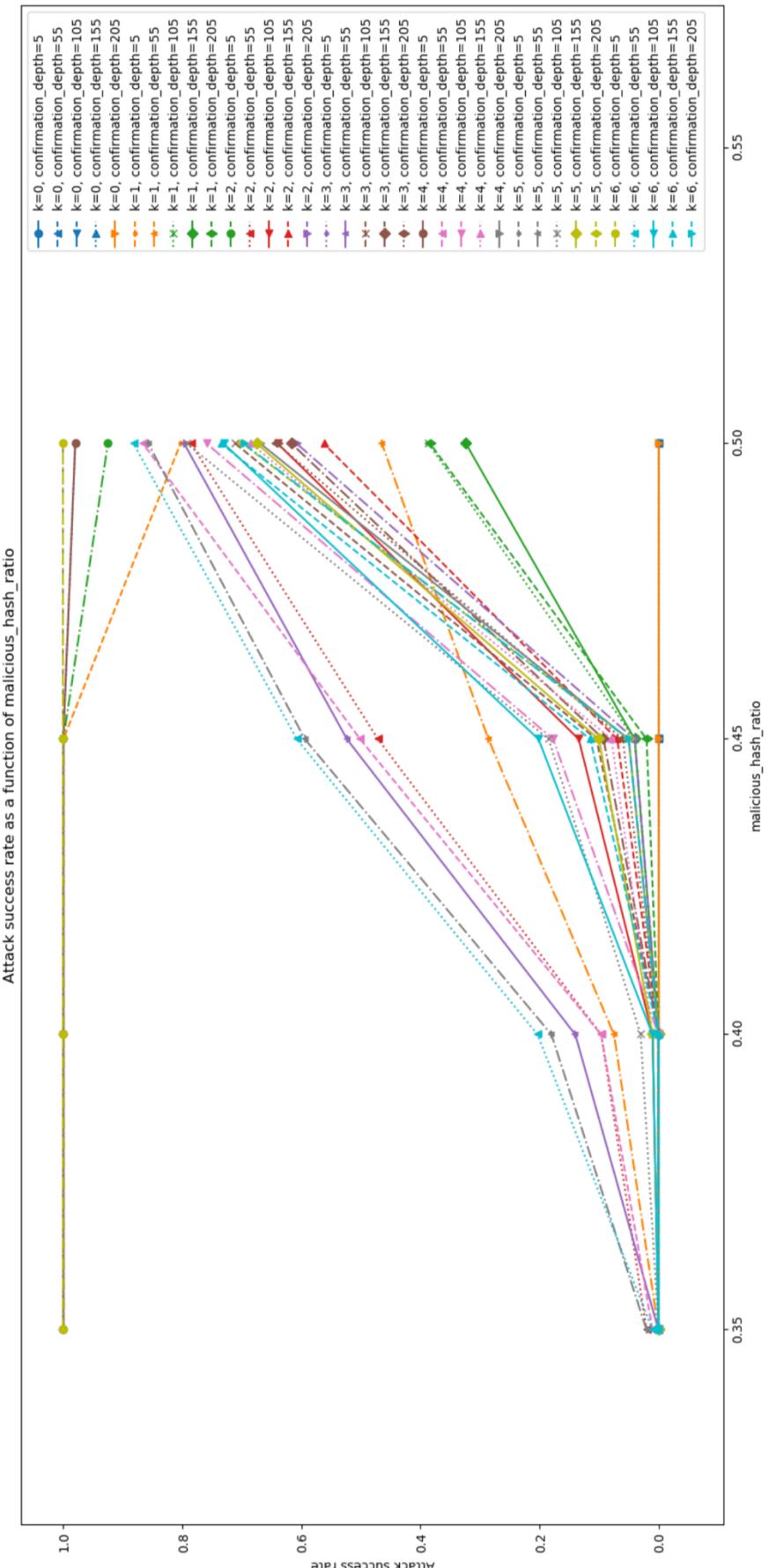


图 3 使用规则 2 时，攻击成功率是关于恶意散列率的函数：



引文

- [1] Y. Sompolinsky and A. Zohary, PHANTOM: A Scalable BlockDAG protocol, 2018.
- [2] Y. Lewenberg, Y. Sompolinsky, and A. Zohary, Inclusive Block Chain Protocols, 2015.

Algorithm 5: UPDATE-MAX-CHAIN($G, block$)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 $block$ - the newly block

Output: G - the graph after updating

```
1  $virtual \leftarrow virtual(G)$ 
2  $parents(virtual) \leftarrow (parents(virtual) \cup \{block\}) \setminus parents(block)$ 
3  $diffpast(virtual) \leftarrow diffpast(virtual) \cup block$ 
4  $order(diffpast(virtual)) \leftarrow \emptyset$ 
5  $oldColoringParent \leftarrow coloringParent(virtual)$ 
6  $coloringParent(virtual) \leftarrow \text{GET-COLORING-PARENT}(virtual)$ 
/* The actual code uses a flat LazySet for the antipast here,
allowing better efficiency: */
7  $diffpast(virtual) \leftarrow \text{GET-ANTIPAST}(coloringParent(virtual))$ 
8  $intersection \leftarrow$ 
    GET-MAX-CHAIN-INTERSECTION( $coloringParent(virtual)$ )
9 for  $curBlock \in oldColoringParent \rightarrow intersection$  do
10   |  $coloringChain(G) \leftarrow coloringChain(G) \setminus \{curBlock\}$ 
11 end
12 for  $curBlock \in coloringParent(virtual) \rightarrow intersection$  do
13   |  $coloringChain(G) \leftarrow coloringChain(G) \cup \{curBlock\}$ 
14 end
15 if  $genesis(G) \notin coloringChain(G)$  then
16   |  $genesis(G) \leftarrow \arg \min\{height(b) : b \in coloringChain(G)\}$ 
17 end
18 return  $G$ 
```

Algorithm 6: GET-MAX-CHAIN-INTERSECTION($G, block$)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 $block$ - a block in G

Output: $intersection$ - the first block on the coloring chain whose tip is $block$
that intersects $coloringChain(G)$

```
1  $intersection \leftarrow block$ 
2 for  $curBlock \in coloringChain(block)$  do
3   | if  $block \in coloringChain(G)$  then
4     |   | return  $curBlock$ 
5   | end
6 end
7 return  $\emptyset$ 
```

Algorithm 7: GET-ANTIPAST($G, block$)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 $block$ - a block in G

Output: $antipast$ - the antipast LazySet of $block$ in G

```
1  $antipast \leftarrow \text{CREATE-LAZYSET}()$ 
2  $intersection \leftarrow$ 
   GET-MAX-CHAIN-INTERSECTION( $\text{coloringParent}(virtual)$ )
3 for  $curBlock \in virtual(G) \rightarrow intersection$  do
4   |  $antipast \leftarrow antipast \cup diffpast(curBlock)$ 
5 end
6 for  $curBlock \in block \rightarrow intersection$  do
7   |  $antipast \leftarrow antipast \setminus diffpast(curBlock)$ 
8 end
9 return  $antipast$ 
```

Algorithm 8: CALC-K-CHAIN(G, k)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 k - the k value

Output: $kChain$ - the k-chain whose tip is the tip of $\text{coloringChain}(G)$

```
1  $kChain \leftarrow \emptyset$ 
2  $depth \leftarrow 0$ 
3 for  $curBlock \in \text{coloringChain}(G)$  do
4   | if  $depth > k$  then
5     |   return  $kChain$ 
6   | end
7   |  $kChain \leftarrow kChain \cup curBlock$ 
8   |  $minHeight(kChain) \leftarrow height(curBlock)$ 
9   |  $depth \leftarrow depth + |\text{blue}(diffpast}(curBlock))|$ 
10 end
11 return  $kChain$ 
```

Algorithm 9: COLORING-RULE_k($G, block$)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 $block$ - a block to color

Output: the color of $block$ according to G

/* In the actual code, $kChain$ is pre-computed once for the
entire coloring process. */

```
1  $kChain \leftarrow \text{CALC-K-CHAIN}(k)$ 
2 for  $curBlock \in \text{coloringChain}(block)$  do
3   if  $block \in kChain$  then
4     | return BLUE
5   end
6   if  $height(block) < minHeight(kChain)$  then
7     | /* Intersection surely occurs only at a lower height, thus
      |       the block is definitely red */ 
8     | return RED
9   end
9 end
```

Algorithm 10: CALC-DIFFPAST-COLOR($G, block$)

Input : G - a block DAG,
 $block$ - a block contained in G

Output: the coloring of $\text{diffpast}(G, block)$ according to G 's coloring rule

```
1  $RULE \leftarrow \text{rule}(G)$ 
2  $blue(\text{diffpast}(block)) \leftarrow \emptyset$ 
3  $red(\text{diffpast}(block)) \leftarrow \emptyset$ 
4  $visited \leftarrow \emptyset$ 
5  $coloringParentAntipast \leftarrow \text{GET-ANTIPAST}(\text{coloringParent}(block))$ 
6  $toVisit \leftarrow \text{CREATE-QUEUE}(\text{parents}(block))$ 
7 while  $toVisit \neq \emptyset$  do
8    $curBlock \leftarrow \text{POP}(toVisit)$ 
9   | /*  $\text{diffpast}(block) = \text{antipast}(\text{coloringParent}(block)) \cap \text{past}(block)$  */
10  | if ( $curBlock \notin visited$ )&( $curBlock \in coloringParentAntipast$ ) then
11    |    $visited \leftarrow visited \cup curBlock$ 
12    |    $toVisit \leftarrow toVisit \cup \text{parents}(curBlock)$ 
13    |   if  $RULE(G|_{block}, curBlock) = \text{BLUE}$  then
14      |     |  $blue(\text{diffpast}(block)) \leftarrow blue(\text{diffpast}(block)) \cup \{curBlock\}$ 
15    |   else
16      |     |  $red(\text{diffpast}(block)) \leftarrow red(\text{diffpast}(block)) \cup \{curBlock\}$ 
17    |   end
18  end
19 end
19 return  $blue(\text{diffpast}(block)), red(\text{diffpast}(block))$ 
```

Algorithm 11: GET-BLUE(G)

Input : $G = (V, E)$ - a PHANTOM block-DAG
Output: $blue(G)$ - the coloring of G

```
1 coloring  $\leftarrow$  CREATE-LAZYSET()
2 for  $curBlock \in virtual(G) \rightarrow genesis(G)$  do
3   | coloring  $\leftarrow$  coloring  $\cup$   $blue(diffpast(curBlock))$ 
4 end
5 return coloring
```

Algorithm 12: GET-DEPTH($G, block$)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 $block$ - a block
Output: $depth$ - the depth of $block$ in G

```
1 if  $block \notin G$  then
2   | return  $-\infty$ 
3 end
4 if  $block \in diffpast(virtual(G))$  then
5   | return 0
6 end
/* From now on, all blocks are behind G's coloring tip */ 
7  $depth \leftarrow 1$ 
8 for  $curBlock \in coloringChain(G)$  do
9   | if  $block \in red(diffpast(block))$  then
10    |   | return 0
11   | end
12   | if  $block \in blue(diffpast(block))$  then
13    |   | return  $depth$ 
14   | end
15   |  $depth \leftarrow depth + |blue(diffpast(block))|$ 
16 end
```

Algorithm 13: SORT-BLOCKS($diffpast$, $last$, $toSort$)

Input : $diffpast$ - a colored diffpast to sort according to,
 $last$ - the block to put last,
 $toSort$ - a set of the blocks to sort

Output: $sorted$ - an ordered list of the blocks in $toSort \cap diffpast$

```
1 sorted ← CREATE-LIST()
2 remainingBlocks ← ( $toSort \setminus \{last\}$ )  $\cap diffpast$ 
   /* SORT is a descending sorting algorithm on the global IDs */
3 blueSorted ← SORT( $remainingBlocks \cap color(diffpast)$ )
4 redSorted ← SORT( $remainingBlocks \setminus blueSorted$ )
5 sorted ← redSorted  $\cup$  blueSorted
6 if  $last \neq \emptyset$  then
7   | sorted ← sorted  $\cup \{last\}$ 
8 end
9 return sorted
```

Algorithm 14: CALC-DIFFPAST-ORDER(G , $block$)

Input : $G = (V, E)$ - a block DAG,
 $block$ - a block with a colored diffpast to order

Output: $order$ - an order on $diffpast(block)$

```
1 if  $coloringParent(block) \neq \emptyset$  then
2   |  $curIndex \leftarrow selfIndex(coloringParent(block))$ 
3 else
4   |  $curIndex \leftarrow 0$ 
5 end
/* The code uses a dictionary as the map */
6  $order \leftarrow \text{CREATE-MAP}()$ 
7  $toOrder \leftarrow \text{CREATE-STACK}()$ 
8  $toOrder \leftarrow toOrder \cup block$ 
9 while  $toOrder \neq \emptyset$  do
10  |  $curBlock \leftarrow \text{POP}(toOrder)$ 
11  | if  $curBlock \notin order$  then
12    |   |  $curParents \leftarrow parents(curBlock) \cap diffpast(block)$ 
13    |   | if  $curParents \subseteq order$  then
14      |     |   | if  $curBlock \neq block$  then
15        |       |     |   |  $order(curBlock) \leftarrow curIndex$ 
16        |       |     |   |  $curIndex \leftarrow curIndex + 1$ 
17      |     |   | end
18    |   | else
19      |     |   |   |  $toOrder \leftarrow (toOrder \cup \{curBlock\}) \cup$ 
20      |     |   |   | SORT-BLOCKS( $diffpast(block)$ ,  $coloringParent(curBlock)$ ,
21      |     |   |   |  $curParents$ )
22  | end
23 end
24 return  $order$ 
```

Algorithm 15: GET-LOCAL-ID($G, block$)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 $block$ - a block

Output: $localID$ - the local ID of $block$ in G

```
1 if  $block \notin G$  then
2   | return  $\infty$ 
3 end
4  $virtual \leftarrow virtual(G)$ 
5 if ( $block \in diffpast(virtual)$ ) and ( $order(diffpast(virtual)) = \emptyset$ ) then
6   | if  $color(diffpast(virtual)) = \emptyset$  then
7     |   |  $color(diffpast(virtual)) \leftarrow CALC\text{-}DIFFPAST\text{-}COLOR(virtual)$ 
8   | end
9   |  $order(diffpast(virtual)) \leftarrow CALC\text{-}DIFFPAST\text{-}ORDER(virtual)$ 
10 end
11 for  $curBlock \in virtual \rightarrow genesis(G)$  do
12   |  $orderMap \leftarrow order(diffpast(curBlock))$ 
13   | if  $block \in orderMap$  then
14     |   | return GET-VALUE( $orderMap, block$ )
15   | end
16 end
```

/* Implemented in code using a ChainMap */

Algorithm 16: GET-MALICIOUS-PARENTS($G, prev$)

Input : $G = (V, E)$ - a PHANTOM block-DAG,
 $prev$ - the previous malicious block

Output: $maliciousParents$ - the parents set of the next malicious block

```
1  $parentAntipast \leftarrow \text{GET-ANTIPAST}(\text{coloringParent}(\text{block}))$ 
2  $maliciousParents \leftarrow \emptyset$ 
3  $visited \leftarrow \emptyset$ 
4  $toVisit \leftarrow \text{CREATE-QUEUE}()$ 
5  $toVisit \leftarrow toVisit \cup \text{parents}(\text{virtual}(G))$ 
6 while  $toVisit \neq \emptyset$  do
7    $curBlock \leftarrow \text{POP}(toVisit)$ 
8    $visited \leftarrow visited \cup \{curBlock\}$ 
9   if  $\text{blueNumber}(curBlock) <$ 
     $\text{blueNumber}(prev)) | ((\text{blueNumber}(curBlock) =$ 
     $\text{blueNumber}(prev)) \& (prev \leq curBlock))$  then
10     $maliciousParents \leftarrow maliciousParents \cup \{curBlock\}$ 
11     $ancestors \leftarrow \text{CREATE-QUEUE}()$ 
12     $ancestors \leftarrow ancestors \cup \text{parents}(curBlock)$ 
13    while  $ancestors \neq \emptyset$  do
14       $curAncestor \leftarrow \text{POP}(ancestors)$ 
15      if  $curAncestor \in parentAntipast$  then
16         $visited \leftarrow visited \cup \{curAncestor\}$ 
17         $maliciousParents \leftarrow maliciousParents \setminus \{curAncestor\}$ 
18         $ancestors \leftarrow ancestors \cup \text{parents}(curAncestor)$ 
19      end
20    end
21  else
22     $| toVisit \leftarrow toVisit \cup \text{parents}(curBlock)$ 
23  end
24 end
25 return  $maliciousParents$ 
```
