Received 1 October 2024; revised 5 November 2024; accepted 10 November 2024. Date of publication 14 November 2024; date of current version 2 December 2024. The review of this article was arranged by Associate Editor Peng Li.

Digital Object Identifier 10.1109/OJCS.2024.3497892

GHOSTForge: A Scalable Consensus Mechanism for DAG-Based Blockchains

MISBAH KHAN [1], SHABNAM KASRA KERMANSHAHI [2], AND JIANKUN HU [1]

[1] University of New South Wales Canberra at the Australian Defence Force Academy Canberra 2617, Australia
[2] School of Systems and Computing Canberra 2612, Australia

CORRESPONDING AUTHOR: JIANKUN HU (e-mail: jiankun.hu@unsw.edu.au).

This work was supported in part by ARC Discovery under Grant DP190103660 and Grant DP200103207 and in part by ARC Linkage under Grant LP180100663.

ABSTRACT Blockchain scalability has long been a critical issue, and Directed Acyclic Graphs (DAGs) offer a promising solution by enabling higher throughput. However, despite their scalability, achieving global convergence or consensus in heterogeneous DAG networks remains a significant challenge. This work, introduces GHOSTForge, building on the Greedy Heaviest-Observed Sub-tree (GHOST) protocol to address these challenges. GHOSTForge incorporates unique coloring and scoring mechanisms alongside stability thresholds and order-locking processes. This protocol addresses the inefficiencies found in existing systems, such as PHANTOM, by offering a more proficient two-level coloring and scoring method that eliminates circular dependencies and enhances scalability. The use of stability thresholds enables the early locking of block orders, reducing computational overhead while maintaining robust security. GHOSTForge's design adapts dynamically to varying network conditions, ensuring quick block order convergence and strong resistance to attacks, such as double-spending. Our experimental results demonstrate that GHOSTForge excels in achieving both computational efficiency and rapid consensus, positioning it as a powerful and scalable solution for decentralized, heterogeneous DAG networks.

INDEX TERMS Blockchains, consensus, direct acyclic graphs, GHOST, scalability, security.

I. INTRODUCTION

DAG-based distributed ledger technologies present a promising solution to the scalability limitations inherent in linear blockchains [1] such as Bitcoin [2] and Ethereum [3]. In these traditional blockchain systems, blocks are added in a strictly linear sequence, where each block references only its immediate predecessor, ensuring easy verification but severely limiting scalability. As transaction volume increases, the throughput of linear blockchains faces challenges to meet growing demands [4].

DAG-based ledgers overcome this bottleneck by leveraging the DAG data structure, represented as $\mathcal{G} = (V, E)$, where V denotes the set of vertices (or blocks), and E represents edges, which are cryptographic links between blocks or transactions. In contrast to linear chains, DAG structures allow multiple blocks or transactions to be confirmed simultaneously, as each new block can reference multiple previous blocks. Protocols such as IOTA [5], PHANTOM [6], SPECTER [7], HASHGRAPH [8], NANO [9], etc. exemplify how DAG-based ledgers facilitate scalability by enabling asynchronous block production, with transactions referencing previous ones to confirm their validity.

Among these protocols, PHANTOM stands out for its close resemblance to traditional linear blockchains, specifically when $K = 0$, where the network behaves similarly to Bitcoin. PHANTOM utilizes a variant of the GHOST protocol, which was originally designed to handle forks in linear blockchains [10], but extended to accommodate the parallel nature of DAGs. In PHANTOM, the GHOST protocol is used to identify the heaviest sub-tree, aiding in block selection during forks, which improves scalability by allowing multiple branches to coexist and merge. The use of GHOST in DAGs allows for more efficient handling of forks, thus facilitating greater transaction throughput.

However, while PHANTOM's GHOST-based approach represents a step forward in terms of scalability, it suffers from computational overhead due to its repetitive coloring, scoring, and ordering calculations along the main chain. This limits its applicability to real-world deployments where computational efficiency is crucial. Moreover, the reliance on a recursive main chain structure introduces circular dependencies between block scoring and coloring, which can hinder performance and consensus finality. Similarly, Conflux [11] achieves consensus by creating a pivot chain and utilizing the GHOST protocol to select the heaviest subtrees. While the experimental results are promising, Conflux inherits the aforementioned issues associated with the GHOST protocol.

To address these limitations, we propose **GHOSTForge**, an enhanced version of the GHOST protocol specifically designed for DAG-based distributed ledgers. GHOSTForge improves scalability, efficiency, and security through the following key contributions:

- *Introduction of Two-Level (Block and DAG) Coloring and Ordering Mechanism:* We propose a novel two-level approach to coloring, ordering, and scoring, utilizing progressive inheritance to increase computational efficiency. This method builds on the previously proposed techniques but eliminates the circular dependency between scoring and coloring that arises from PHANTOM's reliance on the main chain.
- *Stability Threshold and Order Locking:* GHOSTForge introduces a stability threshold and recursive order locking mechanism to ensure global order convergence and consensus. By locking stable prefixes in the DAG as they reach the threshold, we prevent unnecessary recalculations and allow nodes to reach consensus more efficiently.
- *Enhanced Security Against Double Spending Attacks:* We analyze the security of GHOSTForge against multiple double spending attacks, demonstrating that GHOSTForge maintains the same security guarantees as PHANTOM and improves early-stage consensus. This enables GHOSTForge to provide a more secure and efficient network environment, facilitating faster transaction finality.

The rest of the article is organized as follows. Section II reviews related work in DAG-based blockchain protocols. Section III introduces key concepts. Section IV details the GHOSTForge protocol, covering coloring, scoring, and ordering mechanisms. Section V evaluates its efficiency compared to existing methods. Section VI analyzes resilience to double-spending attacks. Section VII discusses node convergence and consensus, and Sections VIII and IX conclude with discussions and future work.

II. RELATED WORK

This section outlines the contributions in the DAG-based blockchain domain and discusses how our work, GHOSTForge, builds upon and improves these previous efforts.

Several early DAG-based ledgers, such as IOTA and NANO, focus on scalability and high throughput. IOTA uses individual transactions as nodes, where each transaction confirms two previous ones [5], while NANO introduces a block-lattice structure with each account maintaining its blockchain [9]. However, both face challenges in achieving global consensus and decentralization, which GHOSTForge addresses through two-level coloring, scoring, and stability thresholds.

Hashgraph uses a gossip protocol to create a consensus graph based on timestamps [8]. Although highly efficient, it requires extensive communication. Similarly, OHIE focuses on scalability but is less suited for DAG-based structures [12]. GHOSTForge enhances these approaches by introducing order-locking mechanisms that reduce communication while ensuring secure order stability across nodes.

Other efforts like Phantasm aim to improve scalability but struggle with computational overhead due to their recursive recalculations [13]. BDLedger focuses on large-scale data recording [14], and CoDAG seeks to minimize storage overhead by merging blockchains [15]. GHOSTForge complements these protocols by ensuring efficient order convergence and scalability.

Protocols such as 3D-DAG [16], FLUID [17], and Conflux [11] aim to improve throughput and processing efficiency, but lack robust mechanisms for ensuring long-term block order stability. Similarly, Byteball [18] and StreamNet [19] focus on linking transactions in a DAG but do not fully address computational complexity or order convergence–issues that GHOSTForge handles with its stability thresholds.

Finally, several newer protocols focus on enhancing consensus efficiency and reducing latency. Blockmania [20], Prism [21], and others like Meshcash [22] target lightweight devices or specific applications such as IoT. GHOSTForge builds upon these by ensuring stability across nodes and minimizing computational overhead, making it suitable for a wide variety of applications.

Recent advancements like JointGraph [23], C-DAG [24], and DEXON [25] introduce new techniques to optimize DAG consensus. RT-DAG [26] and DAG-Block [27] focus on real-time consensus and reducing confirmation times. An efficient consensus model for DAG-based systems was also proposed in [28]. These works provide valuable insights into improving scalability and consensus mechanisms, and GHOSTForge further enhances these areas by incorporating stability thresholds and recursive locking to ensure secure, scalable, and globally consistent consensus.

III. PRELIMINARIES

This article uses several key terms throughout. **Tips** refer to the most recent blocks in the DAG that have not been referenced by other blocks. **Anticones** are blocks neither directly nor indirectly connected to the block under consideration, with the **blue anticone** specifically referring to anticones of blue blocks. In GHOSTForge, newly created **blocks** reference all visible tips, represented as edges in the DAG. **Blue blocks** are considered likely honest and are determined by the **K threshold**, which limits the number of blue anticones a block can have. Blocks exceeding this threshold become **red blocks**. The **blue set** consists of blocks colored blue, while the **red set** contains those outside this set. **Score** refers to the number of blue blocks in the past of a block. Blocks are **topologically ordered** to respect dependencies, as detailed in later sections. GHOSTForge, like the PHANTOM protocol's **K-cluster**, prioritizes the largest blue set to maintain consistency.

Our work extends GHOST from the PHANTOM protocol. For comparison, the original protocol works as follows: (1) Miners create new blocks referencing all tips; (2) A virtual block establishes the Main Chain, ordered by scores from tips to genesis; (3) Blocks are colored based on the K threshold (Ph_{col}); (4) Blue blocks are ordered before red blocks. The greedy chain selection algorithm builds the chain by choosing the highest-scoring predecessors until the genesis block is reached. The blue set is constructed by evaluating blocks in reverse, adding them according to the K threshold to ensure complete DAG ordering (Ph_{ord}). In the remainder of the paper, Ph_{col} and Ph_{ord} will refer to PHANTOM's coloring and ordering techniques.

IV. GHOSTFORGE PROTOCOL

This section formally introduces our proposed protocol, which includes the coloring, scoring, and ordering of blocks, as well as the locking order and stability count mechanisms.

A. BLOCK LEVEL COLORING AND SCORNING

The Algorithm 1 determines the color and score of a block B within a DAG $\mathcal{G}$. If B is the genesis block G, it is assigned a blue color, and its initial score is returned. For other blocks, the algorithm checks for conflicts with blocks already in the blue set B_s. If a conflict with block C_B is detected, C_B is purged, added to the red set R, and B is marked red without further processing.

However, if the block does not conflict with existing blocks in the blue set B_s, the protocol proceeds with the following steps to determine the color and score of block B. The inherited blue set $\mathcal{I}_B$ for B is calculated based on the blue sets of its predecessors P. The parent with the highest score, $P_{\max}$, determines the initial inherited blue set for B, which includes all blue blocks in the past of $P_{\max}$. The algorithm checks if the remaining predecessors (other parents and their past) can be added to $\mathcal{I}_B$ by evaluating their anticones $\mathcal{A}_P$ within $\mathcal{I}_B$ (By calling the helping method in Algorithm 2). Where $\mathcal{A}_P \leftarrow \{b \in \mathcal{G} \mid b \notin \mathcal{I}_B \text{ and } b \in \text{Anticone}(\mathcal{I}_B)\}$ defines $\mathcal{A}_P$ as the set of blocks that are *not connected* (neither ancestors nor descendants) to the blocks in the inherited blue set $\mathcal{I}_B$. If the size of $\mathcal{A}_P$ is less than or equal to the threshold k ($|\mathcal{A}_P| \leq k$), the block P is added to the blue set B_s, the inherited blue set $\mathcal{I}_B$ is extended to include P, and P is removed from the red set R (if it was previously marked red). This evaluation is then recursively applied to the ancestors of P, ensuring consistent processing of all predecessors.

Finally, the algorithm evaluates whether B itself can be added to the blue set by examining its anticone $\mathcal{A}_B$. If $\mathcal{A}_B \leq k$, B is added to B_s, and its score S_B is updated. The scores of all blocks in $\mathcal{I}_B$ are then updated based on their intersection with B_s. The algorithm concludes by returning the color and score of B.

Algorithm 1: Block-Level Coloring and Scoring.

```
1:  Input: $\mathcal{G}$ – DAG, $B$ – block, $R$ – red set, $B_s$ – blue set, $S$ – blue scores, $k$ – $K$-parameter
2:  Output: $color(B)$ – block color, $S(B)$ – block score
3:  if $B = G$ then return (blue, $S[G]$)
4:  end if
5:  $C_B \leftarrow$ DETECT_CONFLICT($\mathcal{G}, B, B_s, m$)
6:  if $C_B \neq$ None then
7:      PURGE_CONF_BLO($\mathcal{G}, B, C_B, B_s, S$)
8:      $R \leftarrow R \cup \{C_B\}$
9:      return (red, 0)
10: end if
11: for all $P \in$ Predecessors($B$) do
12:     Find the parent with the maximum score, $P_{\max}$
13:     $\mathcal{I}_B \leftarrow$ Blueset of $P_{\max}$
14: end for
15: for all $P \in$ Predecessors($B$) do
16:     if $P \neq P_{\max}$ then
17:         K_CHECK($P, \mathcal{I}_B$)
18:     end if
19: end for
20: $\mathcal{A}_B \leftarrow \{b \in \text{Anticone}(\mathcal{G}, B) \mid b \in \mathcal{I}_B\}$
21: if $|\mathcal{A}_B| \leq k$ then
22:     $\mathcal{I}_B \leftarrow \mathcal{I}_B \cup \{B\}$; $B_s \leftarrow B_s \cup \{B\}$; $S_B \leftarrow |\mathcal{I}_B| + 1$
23: else
24:     $S_B \leftarrow |\mathcal{I}_B|$
25: end if
26: UPDATE_BLOCK_SCORES($B, \mathcal{I}_B$)
27: $S[B] \leftarrow S_B$
28: return (color($B$), $S_B$)
```

B. BLOCK LEVEL ORDERING

The block-level ordering Algorithm 3 determines a topological order for all blocks in the past of a given block B, placing B itself at the end of the order. Blocks are ordered primarily based on their scores, with higher-scoring blocks $\succ$ (preceding) lower-scoring ones.

The function `recursive_order` begins by checking if the block B is the genesis block G. If $B = G$, the algorithm returns $[G]$ as the base case, ensuring that the genesis block is always the starting point of the order. For any block $B \neq G$, the algorithm retrieves the list of predecessor blocks P of B using `predecessors`($\mathcal{G}, B$). The predecessor blocks P are sorted based on their scores S or hash values. This sorting ensures that blocks with higher scores are ordered first, and in the event of a tie (same score), blocks are ordered by their hash values, with lower hash values preceding higher ones. This tie-breaking mechanism ensures global convergence by having all miners independently follow the same score and hash-based rules, leading to a consistent and deterministic order without needing communication. This consistency is crucial for maintaining a unified view of the DAG across the network. The algorithm recursively calls `recursive_order`(P_i) for each sorted predecessor P_i, building an ordered list L_B for block B.

Algorithm 2: K_Check: A Helping Method.

```
1:  Input: $P$ – Parent, $\mathcal{I}_B$ – inherited blue set
2:  Output: Updated $\mathcal{I}_B$ and $B_s$
3:  $\mathcal{A}_P \leftarrow \{b \in \mathcal{G} \mid b \notin \mathcal{I}_B$ and $b \in \text{Anticone}(\mathcal{I}_B)\}$
4:  if $|\mathcal{A}_P| \leq k$ then
5:      $\mathcal{I}_B \leftarrow \mathcal{I}_B \cup \{P\}$; $B_s \leftarrow B_s \cup \{P\}$
6:      if $P \in R$ then
7:          $R \leftarrow R \setminus \{P\}$
8:      end if
9:      for all $A \in \text{Predecessors}(P)$ do
10:         if $A \notin \mathcal{I}_B$ then
11:             K_CHECK$(A, \mathcal{I}_B)$
12:         end if
13:     end for
14: else
15:     $R \leftarrow R \cup \{P\}$; $B_s \leftarrow B_s \setminus \{P\}$
16:     for all $A \in \text{Predecessors}(P)$ do
17:         if $A \notin \mathcal{I}_B$ then
18:             K_CHECK$(A, \mathcal{I}_B)$
19:         end if
20:     end for
21: end if
22: return $\mathcal{I}_B, B_s$
```

Algorithm 3: Block-Level Ordering Based on Scores and Hash Values.

```
1:  Input: $\mathcal{G}$ – DAG, $B$ – block, $S$ – blue scores
2:  Output: $L$ – ordered list of blocks
3:  function RECURSIVE_ORDER($N$)
4:      if $B = G$ then                        ▷ Base case: Genesis block
5:          return $[G]$
6:      end if
7:      $P \leftarrow$ predecessors($\mathcal{G}, B$)
8:      $P_{\text{sorted}} \leftarrow$ sorted($P$, key $= \lambda\, x$ :
        $(-S.get(x, 0)$, hash_block$(x)))$
9:      for all $P_i \in P_{\text{sorted}}$ do
10:         $L_B \leftarrow L_B +$ RECURSIVE_ORDER($P_i$)
11:     end for
12:     $L_B \leftarrow L_B + [B]$                ▷ Add current block
13:     return $L_B$
14: end function
15: $L \leftarrow$ RECURSIVE_ORDER($B$)             ▷ Compute the
    block-level ordered list
16: return $L$
```

This recursive approach ensures that all dependencies (past blocks) of B are fully ordered before B itself is added. The block B is then appended to the end of the list L_B. The function finally returns the ordered list L for the block B.

C. INHERIT AND DETERMINE THE ORDER

This Algorithm 4 is designed to establish the final DAG-level order following the creation or reception of block(s) within the network. The method is invoked in Algorithm 5 after DAG-level coloring and leverages Algorithm 3 to ensure the proper sequence of blocks. The primary goal of this method is to efficiently determine whether the DAG-level order can be inherited from the previous maximum tip, T_{prev}, which has already been sorted. There are three checks to determine this inheritance.

First, if the current maximum tip, $T_{\max}$, is found in the previous order list L_{prev}, the method simply inherits the order up to $T_{\max}$ and appends the remaining tips by utilizing Algorithm 3. Second, if $T_{\max}$ is not part of the previous order, the method checks the parents of $T_{\max}$, denoted as $P_{\max}$. If $P_{\max}$ is in L_{prev}, the method inherits the order up to $P_{\max}$, avoiding unnecessary reordering of already sorted blocks. It then applies Algorithm 3 to determine the order of any remaining blocks that were not part of the inherited order but are in the past of $T_{\max}$. Third, if neither of the first two checks are met, the method performs block-level ordering for $T_{\max}$.

Finally, the algorithm checks any remaining tips that were not yet included in the order. These remaining tips are sorted based on their scores and hash values (for tie-breaking), and their blocks are ordered using Algorithm 3. These newly ordered blocks are then appended to the final order. The final order list, L_{new}, is then constructed and returned.

It is important to note that the previous order list, L_{prev}, is initialized by default with the genesis block.

D. DAG-LEVEL COLORING AND ORDERING

The Algorithm 5 extends the principles established in the block-level coloring algorithm (Algorithm 1) to the entire DAG, focusing on all tips rather than individual blocks. While the block-level algorithm processes a single block B based on its past, the DAG-level coloring considers the global view, similar to the hypothetical virtual block concept described in the PHANTOM protocol [6].

Initially, the algorithm retrieves the list of tips $\mathcal{T}$ in the DAG $\mathcal{G}$ and selects the tip $T_{\max}$ with the highest score, using the hash value for tie-breaking. The blue set for $T_{\max}$, denoted as $\mathcal{B}_{\max}$, is computed by including all blue blocks in its past as well as $T_{\max}$ itself. The algorithm recursively checks and adds tips and their ancestors to this set based on the K-parameter condition by leveraging Algorithm 2, specifically, for each tip $t \in \mathcal{T}$ (excluding $T_{\max}$). After processing all tips, the algorithm calls Algorithm 4 to determine the final DAG-level order L_{new} by inheriting the previous order wherever possible and appending newly colored blocks. The purpose of this algorithm is to ensure that the entire DAG is consistently colored and ordered across the network.

Fig. 1 illustrates the process of block-level and DAG-level coloring, scoring, and ordering in a GHOSTForge, with $K = 3$, where new blocks are evaluated based on their connection to the blue set and the K-parameter to determine their color and position in the global order.

E. LOCKING ORDER AND STABILITY COUNT

The purpose of Algorithm 6 is to lock the order of blocks in DAG once a stability threshold is met (e.g., times at t1, t2,

Algorithm 4: Inherit and Determine DAG Order.

```
 1: Input: $\mathcal{G}$ – DAG, $S$ – blue scores, $L_{\text{prev}}$ – previous order, $\mathcal{T}$ – tips list, $T_{\max}$ – max tip, $T_{\text{prev}}$ – previous max tip
 2: Output: $L_{\text{new}}$ – new DAG order
 3: $P \leftarrow \text{Pred}(\mathcal{G}, T_{\max})$   ▷ Find max parent of $T_{\max}$ by score hash
 4: $P_{\max} \leftarrow \text{argmax}_{p \in P}\ (S(p), \text{hash}(p))$
 5: if $T_{\max} \in L_{\text{prev}}$ then   ▷ If Max Tip is already sorted Inherit order
 6:     $L_{\text{in}} \leftarrow L_{\text{prev}}[:\text{index}(T_{\max}) + 1]$
 7: else if $P_{\max} \in L_{\text{prev}}$ then ▷ If Pmax is already sorted Inherit order
 8:     $L_{\text{in}} \leftarrow L_{\text{prev}}[:\text{index}(P_{\max}) + 1]$
 9:     $\mathcal{B}_{\text{rem}} \leftarrow \{b \in \text{past}(\mathcal{G}, T_{\max}) \mid b \notin L_{\text{in}}\}$   ▷ remaining P of Tmax
10:     $L_{\text{rem}} \leftarrow \text{BlockOrder}(\mathcal{G}_{\text{rem}}, T_{\max}, S)$   ▷ Using Algorithm 3
11:     $L_{\text{in}} \leftarrow L_{\text{in}} \cup (L_{\text{rem}} \setminus L_{\text{in}})$ ▷ Extend inherited order
12: else
13:     $L_{\text{in}} \leftarrow \text{BlockOrder}(\mathcal{G}, T_{\max}, S)$   ▷ Using Algorithm 3
14: end if
15: $L_{\text{new}} \leftarrow L_{\text{in}}$   ▷ Add remaining tips
16: $\mathcal{T}_{\text{rem}} \leftarrow \mathcal{T} \setminus L_{\text{new}}$
17: $\mathcal{T}_{\text{sort}} \leftarrow \text{sort}(\mathcal{T}_{\text{rem}}, \text{key} = \lambda t : (-S(t), \text{hash}(t)))$
18: for all $t \in \mathcal{T}_{\text{sort}}$ do
19:     $L_t \leftarrow \text{BlockOrder}(\mathcal{G}, t, S)$   ▷ Using Algorithm 3
20:     $L_{\text{new}} \leftarrow L_{\text{new}} \cup (L_t \setminus L_{\text{new}})$
21: end for
22: $L_{\text{new}} \leftarrow L_{\text{new}} \setminus \mathcal{R}$   ▷ Add red blocks at the end
23: return $L_{\text{new}}$
```

t3 in Fig. 2). The stability threshold is met when a sequence of blocks remains unchanged over a predefined number of iterations, indicating that the order is stable and can be locked. Once the order is locked, subsequent blocks and DAG-level coloring, as explained in Algorithms 1 to 5, continue, focusing only on blocks not part of the locked order until the next stability threshold is reached. This process is illustrated in Fig. 2.

The algorithm begins by checking if the order history, denoted as $\mathcal{H}$, is empty. If it is, the previous order $\mathcal{O}_{\text{prev}}$ is added to the history (recall that the previous order, by default, includes the Genesis block), and the algorithm exits early. If the history is not empty, the algorithm retrieves the last order $\mathcal{O}_{\text{prev}}$ from the history (Previous iteration of coloring and ordering) and compares it with the current order $\mathcal{O}_{\text{curr}}$. The algorithm determines the maximum length of the prefix, denoted as L_{prefix}, that can be compared between the previous and current orders. A prefix in this context refers to the leading sequence of blocks that appears at the beginning of both the previous and current orders. For example, if the orders are $\mathcal{O}_{\text{prev}} = [A, B, C, D, W]$ and $\mathcal{O}_{\text{curr}} = [A, B, C, D, X]$, the

Algorithm 5: DAG-Levl Coloring and Ordering.

```
 1: Input: $\mathcal{G}$ – DAG, $\mathcal{B}$ – blue set, $\mathcal{R}$ – red set
 2: Output: $L_{\text{new}}$ – new DAG order, Updated Coloring
 3: $\mathcal{T} \leftarrow \text{tips}(\mathcal{G})$
 4: $T_{\max} \leftarrow \text{argmax}_{t \in \mathcal{T}}\ (S(t), -\text{hash}(t))$
 5: $\mathcal{B}_{\max} \leftarrow \{b \in \text{past}(\mathcal{G}, T_{\max}) \mid b \in \mathcal{B}\}$ ▷ Get the blue set of $T_{\max}$
 6: $\mathcal{B}_{\max} \leftarrow \mathcal{B}_{\max} \cup \{T_{\max}\}$
 7: for all $t \in \mathcal{T}$ do
 8:     if $t \neq T_{\max}$ then
 9:         K_CHECK$(t, \mathcal{B}_{\max})$   ▷ Algorithm 2
10:     end if
11: end for
12: $L_{\text{new}} \leftarrow$ Call Algorithm 4
13: return $L_{\text{new}}, UpdatedColor$
```

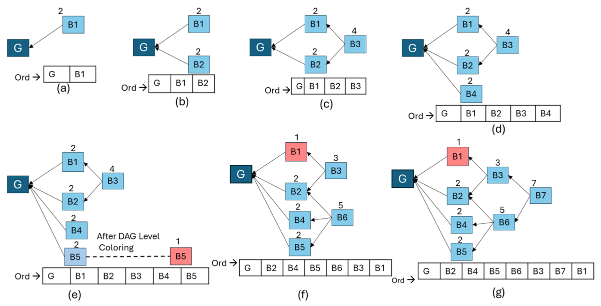


FIGURE 1. **Coloring, Scoring, and Ordering ($K = 3$): (a) Block B1: Block-level: B1 is connected to G with no other blocks, so $K \leq 3$. B1 is colored blue with a score of 2. Block-level order is G, B1. DAG-level: Same as B1. (b) Block B2: Block-level: B2 is connected to G, and $K \leq 3$, so B2 is colored blue with a score of 2. DAG-level: $T_{\max}$ = B1 (lowest hash value); $\mathcal{B}_{\max} = G, B1$. B2 remains blue, and the order becomes (B1's block-level order, B2), i.e., $G, B1, B2$. (c) Block B3: Block-level: Inherits $\mathcal{I}_B = G, B1$. B2 satisfies $K \leq 3$, so B3 is colored blue with a score of 4. DAG-level: Same as B3's block-level order, i.e., $G, B1, B2, B3$. (d) Block B4: Block-level: B4 connects to G and is colored blue with a score of 2. DAG-level: $T_{\max}$ = B3, $\mathcal{B}_{\max} = G, B1, B2, B3$. B4 satisfies $K \leq 3$, so the order becomes (B3's block-level order, B4). (e) Block B5: Block-level: B5 connects to G and is colored blue with a score of 2. DAG-level: $T_{\max}$ = B3, $\mathcal{B}_{\max} = G, B1, B2, B3$ initially, B4 satisfy $K \leq 3$ and added to $\mathcal{B}_{\max}$. B5 fails $K \leq 3$ with 4 anticones and turns red. The order updates to [B3's block-level order, B4, B5(red)]. (f) Block B6: Block-level: Inherits $\mathcal{I}_B = G, B2$ (Lowest hash value). B4 and B5 satisfy $K \leq 3$, so B6 turns blue with a score of 5. DAG-level: $T_{\max}$ = B6, $\mathcal{B}_{\max} = G, B2, B4, B5, B6$. Check remaining tips and their past against $\mathcal{B}_{\max}$, B1 turns red, while B3 remains blue with a score of 3. The order becomes $G, B2, B4, B5, B6, B3, B1$. (g) Block B7: Block-level: Inherits B6's $\mathcal{I}_B$, and B3 is blue against $\mathcal{I}_B$. B7 turns blue with a score of 7. DAG-level: B7 is the only tip, and the order becomes $G, B2, B4, B5, B6, B3, B7, B1$, with red blocks at the end.**

common prefix is $[A, B, C, D]$. If this prefix repeats consistently across multiple iterations (at least k_{stab} times), it is considered a stable prefix, denoted as S_{prefix}.

Next, the algorithm iterates through the prefixes of both orders, updating the stability count C_{stab} for each prefix length i. If the prefixes match, the stability count for that length is incremented. If the stability count reaches or exceeds the stability threshold k_{stab}, the length S_{prefix} is updated to reflect the longest stable prefix. If a stable prefix is identified ($S_{\text{prefix}} > 0$), the algorithm locks the blocks within this prefix

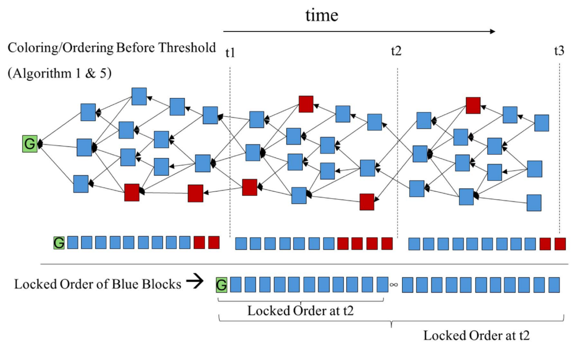

time
Coloring/Ordering Before Threshold (Algorithm 1 & 5)
t1
t2
t3
G
Locked Order of Blue Blocks
Locked Order at t2
Locked Order at t2

FIGURE 2. **GHOSTForge locked order and stability threshold.**

by adding them to the locked blocks set, $\mathcal{L}$. Additionally, all ancestors of the locked blocks are also added to $\mathcal{L}$ to ensure that the locked order is comprehensive and consistent.

Before appending the current order to the order history, the algorithm checks for any conflicts between the current order and the locked order. If a block in the current order conflicts with any block in the locked order (i.e., conflicting hashes), it is removed from the current order. Finally, the non-conflicting current order is appended to the order history $\mathcal{H}$, and the process continues until the next stability threshold is met. This locking mechanism ensures that the network maintains a consistent and stable order of blocks, preventing reordering of previously locked blocks and focusing on new blocks for further processing.

Algorithm 6: Stability Count and Lock Order.

```
1:  Input: $\mathcal{H}$ – order history, $\mathcal{O}_{\text{prev}}$ – previous order,
    $\mathcal{O}_{\text{curr}}$ – current order, $\mathcal{C}_{\text{stab}}$ – stability count, $\mathcal{L}$ –
    locked blocks, $k_{\text{stab}}$ – stability threshold
2:  Output: Updated $\mathcal{H}$
3:  if $\mathcal{H} = \emptyset$ then ▷ Check if the order history is empty
4:      $\mathcal{H}$.append($\mathcal{O}_{\text{prev}}$)
5:      return
6:  end if  ▷ Retrieve the latest order and determine the
    current one
7:  $\mathcal{O}_{\text{prev}} \leftarrow \mathcal{H}[-1]$; $\mathcal{O}_{\text{curr}} \leftarrow \mathcal{O}_{\text{prev}}$
8:  $L_{\text{prefix}} \leftarrow \min(\text{len}(\mathcal{O}_{\text{prev}}), \text{len}(\mathcal{O}_{\text{curr}}))$
9:  for $i \leftarrow 1$ to $L_{\text{prefix}}$ do
10:     if $\mathcal{O}_{\text{curr}}[:i] = \mathcal{O}_{\text{prev}}[:i]$ then
11:         $\mathcal{C}_{\text{stab}}[i] \leftarrow \mathcal{C}_{\text{stab}}.\text{get}(i, 0) + 1$
12:     else
13:         $\mathcal{C}_{\text{stab}}[i] \leftarrow 0$     ▷ Reset stability count if the
            prefix does not match
14:     end if
15:     if $\mathcal{C}_{\text{stab}}[i] \geq k_{\text{stab}}$ then
16:         $S_{\text{prefix}} \leftarrow i$   ▷ Update the longest stable prefix
            length
17:     end if
18: end for
19: if $S_{\text{prefix}} > 0$ then ▷ Lock the longest stable prefix if
    one is found
20:     $\mathcal{L}_{\text{prefix}} \leftarrow \mathcal{O}_{\text{curr}}[:S_{\text{prefix}}]$
21:     for all $b \in \mathcal{L}_{\text{prefix}}$ do
22:         $\mathcal{L}$.add($b$)
23:         $\mathcal{L}$.update(Ancestors($\mathcal{G}, b$))  ▷ Add ancestors of
            locked blocks
24:     end for
25: end if
26: $\mathcal{O}_{\text{non\_conflict}} \leftarrow \emptyset$     ▷ Remove conflicting blocks
    from current order
27: for all $b \in \mathcal{O}_{\text{curr}}$ do
28:     if $b \notin \mathcal{O}_{\text{prev}} \vee \mathcal{O}_{\text{prev}}.\text{index}(b) = \mathcal{O}_{\text{curr}}.\text{index}(b)$
        then
29:         $\mathcal{O}_{\text{non\_conflict}}$.append($b$)
30:     end if
31: end for
32: $\mathcal{O}_{\text{curr}} \leftarrow \mathcal{O}_{\text{non\_conflict}}$
33: $\mathcal{H}$.append($\mathcal{O}_{\text{curr}}$)   ▷ Append updated current order
    to order history
```

V. EVALUATION OF GHOSTFORGE

This section establishes the foundation for GHOSTForge's efficiency. We demonstrate the improvements in both coloring and ordering mechanisms compared to Ph_{ord} and Ph_{col} (refer to Section III). The Ph_{ord} and Ph_{col} methods rely on maintaining a main chain, which becomes computationally intensive as the network scales. GHOSTForge significantly reduces computational overhead by simplifying these processes, introducing the inheritance of color and order at every step without depending on a main chain. Instead, GHOSTForge progressively builds the DAG's order and color, eliminating the need for a main chain. This approach enables GHOSTForge to handle high transaction throughput efficiently, ensuring robust performance even under substantial transaction loads.

Theorem 1: Let $\mathcal{C}_{\text{org}}(G)$ and $\mathcal{C}_{\text{inh}}(G)$ be the computational costs of processing the DAG G in the PHANTOM protocol and the GHOSTForge methods, respectively. Let $P_{\max}$ be the maximum tip under the virtual block, and K_{i} be the number of predecessor tips seen by the $(i+1)$-th node on the main chain $(i = 1, 2, \ldots, n)$, starting from genesis and ending with the virtual block. It holds that:

$$\mathcal{C}_{\text{org}}(G) \geq \mathcal{C}_{\text{inh}}(G) + \sum_{i=2}^{n} O(K_i \log K_i).$$

Proof: Both GHOSTForge and PHANTOM start coloring the DAG G from inheriting the blueset of the $P_{\max}$. However, Ph_{col} will need to establish a main chain, and Ph_{ord} needs to follow this main chain. The overhead of establishing this main chain is $\sum_{i=2}^{n} O(K_i \log K_i)$, where $O(K_i \log K_i)$ is the cost of searching the max tip among K_i. ■

Remarks:

- Although GHOSTForge shares a similar DAG coloring process in terms of inheriting the blue set of the maximum tip under the virtual block's view and expanding the blue set, it does not need to establish the main chain.

TABLE 1. **Experiment Setup and Parameters**

Parameter	Value	Parameter	Value
λ	≈ 9	Network Delay (D)	20-90 ms
Number of Steps	300	Number of Blocks	2000
Duration of Step	30 sec	Total Miners	15

Instead, it expands the blue set by following the topological order of the next largest blue set cluster. As a blue set-expansion-based protocol will result in different DAG-level blue sets by following different block orders of the block coloring process, it is logically sensible to prioritize the next largest blue set cluster. Furthermore, the main-chain-based coloring protocol is confined to the past $P_{\max}$ and the co-tips of $P_{\max}$, and it cannot deal with a batch of incoming interconnected blocks due to a significant delay (e.g., intermediate disconnection, etc.).

- When performing DAG-level ordering, GHOSTForge can inherit a significant portion of the block-level order by retaining the previous blue set order while purging the red blocks into the DAG red set. This boosts the efficiency of the protocol.
- GHOSTForge has utilized the hash value as the priority to address the score tie-break, which ensures a deterministic ordering process, boosting the consensus across the network.
- The above-mentioned properties of GHOSTForge assign it the capacity to lock in the prefix sub-DAG: i.e. when all nodes share the same prefix sub-DAG for a while, it will form a stabilized prefix order for the growing DAG. Subsequently, this sub-DAG can be locked without being involved in the further coloring and ordering process, leading to further efficiency enhancement.

To support Theorem 1, we conducted an experiment comparing the execution time of the PHANTOM and GHOSTForge protocols. The experimental setup and parameters are the same as those mentioned in Table 1 (Section VII), with the exception of the number of blocks, which is set to 400in this experiment to better highlight the fluctuations in execution time for GHOSTForge.

Fig. 3 compares the execution time of PHANTOM and GHOSTForge. The x-axis represents the number of blocks, while the y-axis shows the execution time required for respective protocol operations (e.g., block coloring, scoring, and ordering).

The PHANTOM execution time (blue curve) begins relatively low but increases steadily as the DAG grows. This increase is due to the computational overhead required to recalculate the main chain, re-score, re-color, and re-order blocks each time the DAG expands. As more blocks are added, PHANTOM's execution time scales significantly, highlighting its increasing computational cost as the DAG size increases.

In contrast, GHOSTForge execution time (orange curve) remains much more stable and lower throughout the process. Although there are fluctuations, these can be attributed to the nature of the order-locking mechanism. When the order

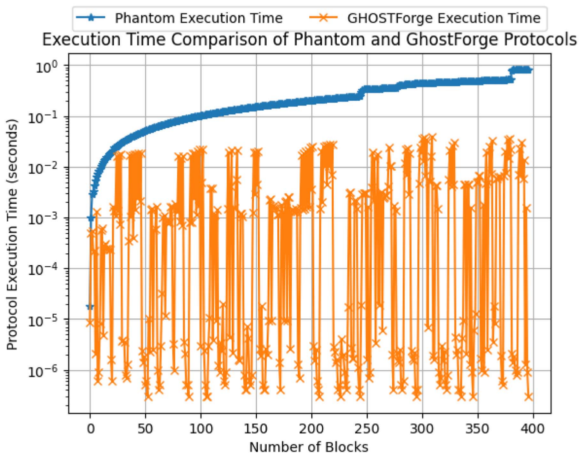


FIGURE 3. **Execution time comparison of PHANTOM and GHOSTForge protocols.**

is locked, fewer blocks require processing, leading to lower execution times. However, as the protocol approaches the locking point where many blocks are still being processed before the final order is locked, we see spikes in the execution time. These spikes occur because more blocks need to be evaluated and ordered just before the stability threshold is reached. After the order is locked, the number of blocks requiring further recalculations is significantly reduced, leading to a decrease in execution time. This fluctuation highlights the efficiency of GHOSTForge in managing block order with minimal computational overhead after the locking point, making the protocol highly scalable as the DAG grows larger. Overall, GHOSTForge's execution times remain consistently lower than PHANTOM's. This behavior emphasizes the protocol's efficiency in locking block order and limiting the need for frequent recalculations, even as the DAG grows.

A. *RESOLVING CIRCULAR DEPENDENCIES IN SCORING AND COLORING*

In the proposed method, we address the circular dependency found in the PHANTOM protocol by separating the scoring and coloring processes into two distinct stages: block-level and DAG-level. In PHANTOM, the score of a block is determined by the number of blue blocks in its past, and the main chain is greedily constructed based on these scores. However, since the main chain is also the basis for determining a block's color, this creates a circular dependency: the main chain is needed for coloring, yet scoring depends on colors that require the main chain. To resolve this, our method first handles scoring and coloring at the block level. Each block B computes its score based on the blue set B_s inherited from its most connected predecessor $P_{\max}$, allowing the score and color to be determined using local information independently of the final DAG structure or the main chain. After all blocks have been initially colored and scored, DAG-level operations are performed, during which the global structure of the DAG is updated, and the scores are refined based on the final

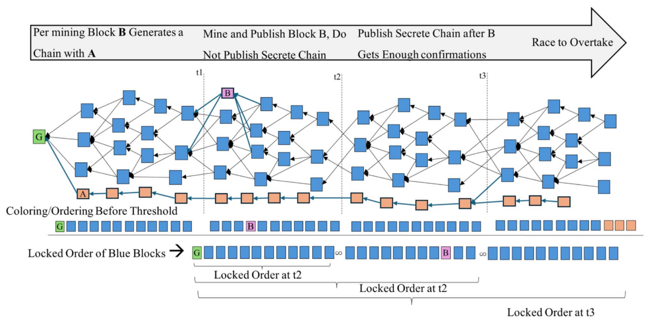

Per mining Block B Generates a Chain with A
Mine and Publish Block B, Do Not Publish Secrete Chain
Publish Secrete Chain after B Gets Enough confirmations
Race to Overtake
t1
t2
t3
Coloring/Ordering Before Threshold
Locked Order of Blue Blocks
Locked Order at t2
Locked Order at t2
Locked Order at t3

FIGURE 4. GHOSTForge resilience to double spending attack: Parasite/hidden chain attack.

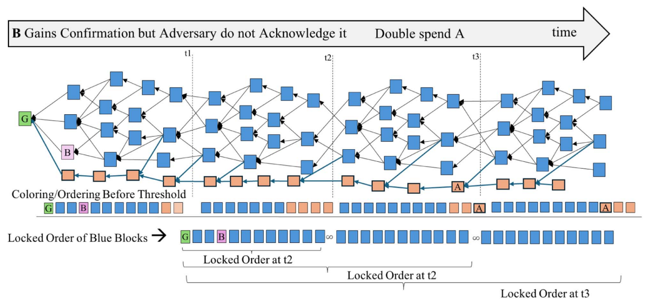

B Gains Confirmation but Adversary do not Acknowledge it
Double spend A
time
t1
t2
t3
Coloring/Ordering Before Threshold
Locked Order of Blue Blocks
Locked Order at t2
Locked Order at t2
Locked Order at t3

FIGURE 5. GHOSTForge resilience to censorship attack.

coloring of the blocks. By decoupling block-level and DAG-level handling, we break the circular dependency, providing a consistent and coherent approach to block scoring and coloring, which avoids the limitations of PHANTOM's reliance on a pre-established main chain.

VI. RESILIENCE TO DOUBLE SPENDING

The GhostForge protocol provides robust security against multiple kinds of double-spending attacks by leveraging a sophisticated locking mechanism within its DAG-based structure. Figs. 4 and 5 illustrate how double-spending attacks, as defined in the SPECTRE protocol, are handled.

GhostForge not only maintains the security levels established by SPECTRE [7] and PHANTOM [6], but it also provides early-stage protection through its locking mechanism. Additionally, GhostForge is resilient to hybrid attacks as defined in [29], as well as balance attacks as described in [30]. This Balance attack is discussed using the GHOST protocol used in the Ethereum blockchain. Furthermore, GhostForge can effectively counter attacks discussed in IOTA Tangle [5] and [13], such as the parasite attack and split attack. By design, the parasite attack is similar to the double-spending attack depicted in Fig. 4, while the split attack is comparable to a balanced attack.

The following sections discuss three types of double-spending attacks: the parasite chain or hidden chain scenario in Section A, the censorship attack in Section B, and the balanced attack in Section C. Finally, Section D explains GhostForge's resilience to all these attacks.

A. DOUBLE SPENDING ATTACK 1: PARASITE/HIDDEN CHAIN ATTACK

The following outlines the key steps involved in a Parasite/Hidden Chain double spending attack (Fig. 4):

- *Pre-mining and Secret Chain Construction:* The attacker begins by pre-mining block *A* and constructing a secret chain that diverges from the honest network's chain. This chain remains hidden while the honest network builds on its visible chain.
- *Broadcasting Block B:* After a period, the attacker creates and broadcasts block *B*, which becomes part of the honest DAG and starts gaining confirmations.
- *Confirmation and Stability:* The attacker waits until block *B* is confirmed by the honest network and becomes part of the stable, heaviest blue cluster.
- *Revealing the Secret Chain:* Once block *B* has received enough confirmations (e.g., accepted by a merchant), the attacker reveals the secret chain that starts from block *A*, which directly conflicts with block *B*, aiming to execute a double spend.

B. DOUBLE SPENDING ATTACK 2: CENSORSHIP ATTACK

The following outlines the key steps involved in a Censorship Double Spending Attack (Fig. 5):

- *Attack Overview:* Block *B* is initially generated and confirmed by the honest network, becoming part of the heaviest blue cluster. However, the adversary ignores block *B* and begins building on an alternative branch, deliberately disregarding its existence.
- *Censorship Attempt:* The adversary continues mining blocks on this alternative branch, attempting to construct a chain that appears more appealing (i.e., a heavier blue cluster). These adversarial blocks are marked in orange, representing the adversary's effort to push an alternative history that excludes block *B*.
- *Double Spend Attempt (Block A):* After some time, the adversary introduces block *A* into their secret chain, which includes a conflicting transaction aimed at performing a double spend. By broadcasting this alternative chain, the adversary hopes the network will accept it over the existing chain, censor block *B*, and validate the conflicting transaction in block *A*.

C. BALANCE/SPLIT ATTACK OVERVIEW

The Balance attack is a sophisticated form of attack that targets proof-of-work blockchains, particularly those like Bitcoin and Ethereum. However, we will discuss this briefly due to its relevance to the GHOST protocol. In a Balance attack, the attacker partitions the network into subgroups, each with roughly equal mining power. The attacker then delays the communication between these subgroups, preventing them from properly sharing information. This delay allows the attacker to mine blocks in one subgroup while keeping another subgroup isolated. After a sufficient number of blocks have been mined, the attacker reconnects the subgroups and attempts to introduce a longer chain that includes the attacker's blocks, potentially rewriting the blockchain and enabling double-spending.

D. GHOSTFORGE RESILIENCE MECHANISMS

GhostForge is inherently resilient to the aforementioned attacks. The following sections provide detailed explanations of the mechanisms that protect against each type of attack.

1) LOCKED ORDER MECHANISM

The GhostForge protocol introduces a stability threshold that locks the order of blocks once they reach a certain level of confirmation. This is visually represented at time steps t_1, t_2, and t_3 in the diagrams for both attacks (Figs. 4 and 5), where the order of the blue blocks, including block B, is locked. This locking mechanism (Algorithm 6: lines 26-32) ensures that once block B and other honest blocks are sufficiently confirmed, their order is preserved within the network, making it impossible for the attacker to replace block B and its subsequent blocks with a secret chain.

Additionally, the probability P_{success} of a successful double-spending attack decreases exponentially with the number of blocks m built on block B. Given the hashing power distribution between the attacker (α) and the honest network ($1-\alpha$) [6], this probability can be expressed as $P_{\text{success}} = (\frac{\alpha}{1-\alpha})^m$, where α represents the attacker's hashing power and $1-\alpha$ represents the honest network's hashing power. As m increases, the probability of a successful attack diminishes, reinforcing the security of GhostForge.

2) EARLY-STAGE PROTECTION

GHOSTForge ordering mechanism is effective even in the initial stages of block confirmation. By locking the order of blocks like B early, the protocol prevents an attacker from gaining an advantage with a secret chain starting from block A, even before the network fully converges. This early protection reduces the window of opportunity for the attacker to execute a double-spending or censorship attack, as the protocol quickly locks in the legitimate transaction order, ensuring that any conflicting transactions introduced by the attacker in the secret chain are invalidated.

3) DOMINANCE OF THE HONEST CLUSTER

A key inherent security feature of GhostForge is that the heaviest cluster of blue blocks, which includes block B, will always belong to the honest network. This is because the honest network collectively has more hashing power than any malicious miner or attacker. As a result, even if the attacker attempts to publish a secret chain starting from block A after block B has received confirmations, the honest chain's cumulative hashing power ensures that the attacker's chain cannot overtake it. The honest network will always generate a heavier blue cluster, thereby preserving the integrity of the transaction history and securing block B against replacement.

4) NETWORK PARTITIONING AND BALANCE ATTACK RESILIENCE

GHOSTForge is resilient to balance attacks due to its ability to secure the locked order of confirmed blocks in the honest chain. The probability of maintaining a successful network partition, allowing the attacker to gain an advantage, follows a binomial distribution, as modeled similarly to (17) in [29]:

$$P_{\text{partition}} = \sum_{i=1}^{sub} \binom{sub}{i} \alpha^i (1-\alpha)^{sub-i} \tag{1}$$

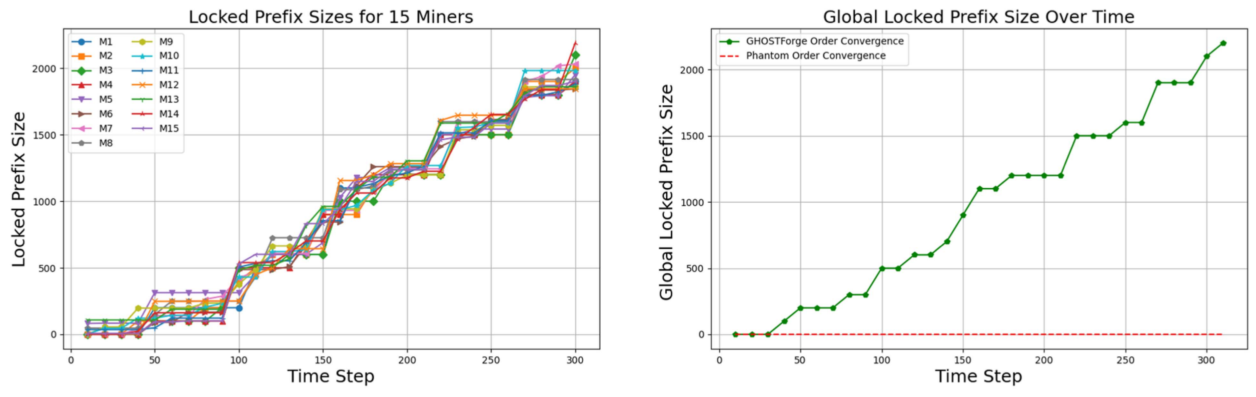
where sub is the number of subgroups the network is partitioned, and α is the fraction of the network's hashing power controlled by the attacker. This equation calculates the likelihood of the attacker controlling i out of sub subgroups.

GHOSTForge addresses these attacks by ensuring the locked order of confirmed blocks, which become irreversible once they reach a stability threshold. Even if the attacker controls part of the network during the partition, GHOSTForge maintains the correct order of the chain after reconnection, rendering the attack ineffective. As discussed in Section IV-C of [30], the ability of an attacker to reverse or reorganize the honest chain diminishes as the chain length increases. GHOSTForge extends this principle by utilizing the locked order property, where blocks, once stabilized, can no longer be reorganized. Any conflicting blocks introduced by the attacker after reconnection will be rejected. Additionally, the stability threshold ensures the finality of the block order in the honest chain, making it impossible for the attacker to disrupt the main chain with privately mined blocks during the partition.

VII. NODE CONVERGENCE AND CONSENSUS

We introduce a mechanism based on stability thresholds and order locking to ensure node convergence and consensus. In a DAG-based network, the current state of miners is heterogeneous due to network delays, resulting in different local views of the DAG. This leads to variations in block coloring, scoring, and ultimately block order across different nodes. The PHANTOM protocol acknowledges that its design is agnostic to multi-miner setups (see Section 3.4 of PHANTOM) and IOTA's consensus evolved through milestones and newer versions to address the challenge of achieving global consensus in a heterogeneous DAG structure.

In contrast, GHOSTForge introduces a stability threshold and a recursive order-locking mechanism to achieve node convergence more efficiently. The stability threshold in GHOSTForge operates under the assumption that a majority of the network's hashing power is controlled by honest miners, i.e., $1-\alpha > 50\%$ (Property 1in PHANTOM). Over time, as blocks propagate across the network, miners will eventually reach a consistent view of the DAG. In PHANTOM, recursive main chain construction continually updates the order, assuming that the order eventually converges. However, this approach is computationally expensive and lacks a defined point where consensus is assured. In GHOSTForge, by locking the order at specific points through the stability threshold,



(a) Miners Locked Prefix

(b) Global Convergence/ Consensus

FIGURE 6. Order convergence and global consensus.

we ensure that the block order converges across all nodes without needing to recursively revisit the genesis block. This allows for early-stage consensus by locking the common prefix, which significantly reduces computational overhead.

Property 1 (Convergence of Block Order via Stability Threshold): Let $\mathcal{N}$ be the set of miners with local views of the DAG, and $\mathcal{O}_i$ the block order for miner $i \in \mathcal{N}$. Let $\mathcal{C}_{\text{stab}}$ be the stability count for each prefix and k_{stab} the threshold to lock a common prefix. Assuming honest miners control the majority of the hashing power ($1 - \alpha > 50\%$), the following holds: If a prefix remains unchanged for k_{stab} iterations across all miners, it is locked as stable. After locking, the remaining blocks are processed independently. With this stability threshold, the block orders $\mathcal{O}_i$ and $\mathcal{O}_j$ for any miners $i, j \in \mathcal{N}$ will converge, ensuring:

$$\mathcal{O}_i = \mathcal{O}_j \quad \text{for all} \quad i, j \in \mathcal{N}.$$

To substantiate the property of block order convergence, we provide empirical evidence through graphs illustrating the role of the stability threshold in achieving consensus across miners. The experiment simulates 15 miners over 300 time steps, with each step lasting 30 seconds. Each miner has a 60% chance of creating a block per step to mimic real network scenarios. Network and reception delays are set to 20 ms-90 ms.[1] The total expected number of blocks generated during the experiment is approximately 2000 (Table 1). As seen in Fig. 6(a), the locked prefix size increases consistently over time as the blocks propagate and are processed by different miners. Once the stability count reaches the threshold k_{stab} (for this experiment $k_{\text{stab}} = 10$), the prefix is locked, preventing further reordering of these blocks. This steady growth of the locked prefix across miners visually demonstrates the efficiency of GHOSTForge in stabilizing the block order, even in a network with inherent delays and heterogeneous views.

Fig. 6(b) (Green line) further confirms that GHOSTForge achieves global consensus by comparing the locked prefix size across all miners. The graph reveals that the locked prefix size across miners converges as the stability threshold is met, and the difference between their local views diminishes. This convergence is the result of the recursive order-locking mechanism employed by GHOSTForge, which allows miners to lock common prefixes early, avoiding the need for continuous reevaluation of the entire DAG.

PHANTOM, on the other hand (Red line), shows no such convergence in the graph. One significant reason for this lack of convergence is PHANTOM's reliance on recalculating the main chain, coloring, and ordering every time the DAG expands. PHANTOM's method for resolving ties in the block order–by arbitrarily selecting a block–introduces substantial variability. Since different miners may select different blocks for tie-breaking, based on local perspectives and selected blue clusters, the overall block order keeps changing as the DAG grows. These variations in local order ultimately result in different global orders across miners.

One key addition that GHOSTForge introduces to overcome this issue is tie-breaking based on hash values rather than arbitrary selection. By enforcing a rule to always select the block with the lowest hash value in the event of a tie, GHOSTForge ensures that all miners choose the same block, even without further communication. This deterministic selection eliminates the risk of inconsistent block orders and guarantees that all miners can lock onto the same prefix early. As a result, GHOSTForge achieves a more stable and consistent global order.

The results highlighted in these graphs support the property of block order convergence. When the stability threshold k_{stab} is met, the block order between miners will converge, validating the theoretical framework and practical application of GHOSTForge in heterogeneous network environments. By locking stable prefixes early, GHOSTForge conserves computational resources while maintaining a reliable and efficient consensus process. This behavior contrasts with recursive main chain updates, which require continuous reevaluation and do not offer early consensus locking.

[1]Please note that delay was chosen based on real-world conditions derived from ping responses to servers across different network environments, covering most practical scenarios.

VIII. DISCUSSION AND FUTURE WORK

The stability count and threshold in GHOSTForge are critical parameters for ensuring block order convergence. These parameters must be carefully tuned based on network conditions and block generation rate for optimal performance.

Let D represent the network delay, which includes both the propagation and reception delays. In high-latency networks where D is large, a higher stability threshold k_{stab} is required to ensure that all nodes receive and process the same information. This prevents premature locking of the block order by any miner before others have fully synchronized their views of the DAG. Formally, we can express this condition as $k_{\text{stab}} > f(D)$. where $f(D)$ is a function that captures the effects of network delay on the stability threshold. As D increases, $f(D)$ should also increase to account for the longer delays in communication between miners.

The block generation rate λ, defined as the expected number of blocks generated per time step, also significantly affects the stability count. In cases where the block generation rate is high, the DAG grows quickly, and the system can stabilize with a lower stability threshold. Specifically, the time for the DAG to reach a stable state, t_{stab}, is inversely proportional to the block generation rate $t_{\text{stab}} \propto \frac{1}{\lambda}$. Thus, a higher λ results in a faster stabilization of the block order, allowing blocks to be locked earlier. Conversely, when λ is low, a higher stability threshold may be needed to ensure miners converge on the same block order.

Another important consideration is the trade-off between security and efficiency. A higher stability count k_{stab} enhances security by reducing the likelihood of inconsistencies in the block order. This is particularly important in adversarial environments, where the risk of conflicting blocks or malicious miners is higher. However, increasing k_{stab} also introduces delays in finalizing blocks, reducing the overall throughput of the system. The goal is to find an optimal balance between security and performance, expressed as: Security $\propto k_{\text{stab}}, \quad$ Efficiency $\propto \frac{1}{k_{\text{stab}}}$. By adjusting k_{stab}, we can tune the system based on the specific network conditions, ensuring that it operates both securely and efficiently.

As a future work, it will be interesting to explore dynamic stability thresholds $k_{\text{stab}}(t)$, which adapt in real-time based on changing network conditions, such as fluctuations in latency or the block generation rate. This will allow the system to maintain high performance even as the network evolves. Another open research question is how to formalize the trade-offs between security and scalability by developing models that capture the interaction between network delays, block generation rates, and the stability threshold. These models will guide the development of adaptive strategies that balance security with performance, ensuring that GHOSTForge remains robust and scalable in diverse network environments.

IX. CONCLUSION

GHOSTForge offers a scalable, efficient, and secure consensus mechanism for DAG-based systems by introducing stability thresholds and order-locking mechanisms. Unlike recursive protocols such as PHANTOM, which continuously updates the block order, GHOSTForge stabilizes the order early, significantly reducing computational overhead. The protocol's resilience to double-spending attacks and its efficient handling of block order convergence makes it a robust solution for networks with heterogeneous views and varying latencies. By locking stable prefixes early, GHOSTForge conserves computational resources, ensuring that all nodes in the network converge to the same block order without the need for ongoing recalculations. Future work will explore dynamic stability thresholds to further improve performance under fluctuating network conditions. The experimental results validate GHOSTForge's ability to achieve fast, secure consensus, positioning it as a robust alternative to current DAG-based consensus mechanisms.

REFERENCES

[1] D. Yang, C. Long, H. Xu, and S. Peng, "A review on scalability of blockchain," in *Proc. 2nd Int. Conf. Blockchain Technol.*, 2020, pp. 1–6.
[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Satoshi Nakamoto*, 2008.
[3] V. Buterin et al., "Ethereum white paper," *GitHub Repository*, vol. 1, pp. 22–23, 2013.
[4] M. Kaur, M. Z. Khan, S. Gupta, A. Noorwali, C. Chakraborty, and S. K. Pani, "MBCP: Performance analysis of large scale mainstream blockchain consensus protocols," *IEEE Access*, vol. 9, pp. 80931–80944, 2021.
[5] W. F. Silvano and R. Marcelino, "Iota tangle: A cryptocurrency to communicate Internet-of-Things data," *Future Gener. Comput. Syst.*, vol. 112, pp. 307–319, 2020.
[6] Y. Sompolinsky, S. Wyborski, and A. Zohar, "PHANTOM GHOSTDAG: a scalable generalization of Nakamoto consensus: September 2, 2021," in *Proc. 3rd ACM Conf. Adv. Financ. Technol.*, 2021, pp. 57–70.
[7] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "SPECTRE: A fast and scalable cryptocurrency protocol," Cryptol. ePrint Arch., 2016.
[8] L. Baird, "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance," Swirlds, Tech. Rep. SWIRLDS-TR-2016-01, 2016.
[9] C. LeMahieu, "Nano: A feeless distributed cryptocurrency network," 2018. [Online]. Available: https://nano.org/en/whitepaper
[10] A. Kiayias and G. Panagiotakos, "On trees, chains and fast transactions in the blockchain," in *Proc. 5th Int. Conf. Cryptol. Inf. Secur. Latin Amer.*, Havana, Cuba, Springer, 2019, pp. 327–351.
[11] C. Li, P. Li, D. Zhou, W. Xu, F. Long, and A. Yao, "Scaling nakamoto consensus to thousands of transactions per second," 2018, *arXiv:1805.03870*.
[12] H. Yu, I. Nikolić, R. Hou, and P. Saxena, "OHIE: Blockchain scaling made simple," in *Proc. 2020 IEEE Symp. Secur. Privacy (SP)*, IEEE, 2020, pp. 90–105.
[13] Z. Zhang et al., "Phantasm: Adaptive scalable mining toward stable blockdag," *IEEE Trans. Serv. Comput.*, vol. 17, no. 3, pp. 1084–1096, May/Jun. 2024.
[14] G. Huang et al., "BDLedger: A scalable distributed ledger for large-scale data recording," in *Proc. 3rd Int. Conf. Blockchain Trustworthy Syst.*, Guangzhou, China, Springer, 2021, pp. 87–100.
[15] S. Yang, Z. Chen, L. Cui, M. Xu, Z. Ming, and K. Xu, "CoDAG: An efficient and compacted DAG-based blockchain protocol," in *Proc. 2019 IEEE Int. Conf. Blockchain*, 2019, pp. 314–318.
[16] J. Zou, Z. Dong, A. Shao, P. Zhuang, W. Li, and A. Y. Zomaya, "3D-DAG: A high performance DAG network with eventual consistency and finality," in *Proc. 1st IEEE Int. Conf. Hot Inf.-Centric Netw.*, 2018, pp. 262–263.
[17] J. Ni, J. Xiao, S. Zhang, B. Li, B. Li, and H. Jin, "FLUID: Towards efficient continuous transaction processing in DAG-based blockchains," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12679–12692, Dec. 2023.

[18] A. Churyumov, “Byteball: A decentralized system for storage and transfer of value,” 2016. [Online]. Available: https://byteball.org/Byteball.pdf

[19] Z. Yin et al., “StreamNet: A DAG system with streaming graph computing,” in *Proc. Future Technol. Conf.*, 2021, pp. 499–522.

[20] G. Danezis and D. Hrycyszyn, “Blockmania: From block DAGs to consensus,” 2018, *arXiv:1809.01620*.

[21] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, “Prism: Deconstructing the blockchain to approach physical limits,” in *Proc. 2019 ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 585–602.

[22] I. Bentov, P. Hubáček, T. Moran, and A. Nadler, “Tortoise and hares consensus: The meshcash framework for incentive-compatible, scalable cryptocurrencies,” in *Proc. 5th Int. Symp. Cyber Secur. Cryptogr. Mach. Learn.*, Springer, 2021, pp. 114–127.

[23] F. Xiang, W. Huaimin, S. Peichang, O. Xue, and Z. Xunhui, “Jointgraph: A DAG-based efficient consensus algorithm for consortium blockchains,” *Softw.: Pract. Exp.*, vol. 51, no. 10, pp. 1987–1999, 2021.

[24] Z. Zhang, D. Zhu, and B. Mi, “C-DAG: Community-assisted DAG mechanism with high throughput and eventual consistency,” in *Proc. 15th Int. Conf. Wireless Algorithms Syst. Appl.*, Qingdao, China, Springer, 2020, pp. 113–121.

[25] T.-Y. Chen, W.-N. Huang, P.-C. Kuo, H. Chung, and T.-W. Chao, “DEXON: A highly scalable, decentralized DAG-based consensus algorithm,” 2018, *arXiv:1811.07525*.

[26] G. Liao, H. Ding, C. Zhong, and Y. Lei, “RT-DAG: A DAG-based blockchain supporting real-time transactions,” *IEEE Internet Things J.*, vol. 11, no. 20, pp. 32759–32772, Oct. 2024.

[27] N. Qi, Y. Yuan, and F.-Y. Wang, “DAG-BLOCK: A novel architecture for scaling blockchain-enabled cryptocurrencies,” *IEEE Trans. Computat. Social Syst.*, vol. 11, no. 1, pp. 378–388, Feb. 2024.

[28] L. Li, D. Huang, and C. Zhang, “An efficient DAG blockchain architecture for IoT,” *IEEE Internet Things J.*, vol. 10, no. 2, pp. 1286–1296, Jan. 2023.

[29] L. Kovalchuk, R. Oliynykov, Y. Bespalov, and M. Rodinko, “Comparative analysis of consensus algorithms using a directed acyclic graph instead of a blockchain, and the construction of security estimates of spectre protocol against double spend attack,” in *Information Security Technologies in the Decentralized Distributed Networks*. Berlin, Germany: Springer, 2022, pp. 203–224.

[30] C. Natoli, P. Ekparinya, G. Jourjon, and V. Gramoli, “Blockchain double spending with low mining power and network delays,” *Distrib. Ledger Technol.: Res. Pract.*, 2024.

MISBAH KHAN received the B.S. and M.S. degrees in software engineering from the University of Management and Technology, Lahore, Pakistan, in 2017 and 2019, respectively. She was a blockchain consultant and IT Business Development Executive for three years. She is currently a Ph.D. degree Researcher with the University of New South Wales (UNSW) Canberra, Australian Defence Force Academy, ACT, Australia. Her research focuses on the applications of blockchain technology, with an emphasis on enhancing security and scalability, particularly in IoT systems.

SHABNAM KASRA KERMANSHAHI received the Ph.D. degree in security sciences, with a focus on applied cryptography from Monash University, Melbourne, Australia. She has a extensive experience in the field of cybersecurity, was also a Postdoctoral Research Fellow with CSIRO’s Data61, Lecturer with RMIT University, Melbourne, Australia, and a Senior Lecturer with UNSW Canberra. She is currently a Senior Lecturer specializing in cybersecurity with the University of New South Wales (UNSW) Canberra, Australia. Her research interests include the areas of cryptography, cybersecurity, information privacy, Automotive Cyber Security, and blockchain. Dr. Kasra was the recipient of the International Association of Geomagnetism and Aeronomy Young Scientist Award for Excellence in 2008, and the IEEE Electromagnetic Compatibility Society Best Symposium Paper Award in 2011.

JIANKUN HU is currently a Full Professor with the School of Engineering and Information Technology, University of New South Wales, Canberra, Australia. He is also an Invited Expert with Australia Attorney-Generals Office assisting in the draft of the Australia National Identity Management Policy. He has authored or coauthored many articles in top venues. His research interests include the field of cybersecurity covering intrusion detection, sensor key management, and biometrics authentication. He was on the Editorial Board of up to seven international journals. He was the recipient of the ten Australian Research Council (ARC) Grants and was also on the prestigious Panel of Mathematics, Information, and Computing Sciences (MIC), and ARC ERA (The Excellence in Research for Australia) Evaluation Committee.