PHANTOM:
AN EFFICIENT IMPLEMENTATION

AVIV YAISH

School of Engineering and Computer Science,
The Hebrew University of Jerusalem, Israel

Abstract. PHANTOM[1] is a new protocol for distributed transaction confirmation designed to be both scalable and secure, one-upping the Blockchain protocol and its famous scalability-security trade-off. Presented here is an efficient implementation of a variant of the protocol, and of a simulation framework designed to test block-DAG protocols.

Contents

1. Implementation 1
2. Simulation framework 9
3. Results & conclusions 10
4. Future work 11
Appendices 11
Acknowledgment 11
References 15

1. Implementation

Presented here is a greedy implementation of PHANTOM, meaning that each block "inherits" the coloring of the parent block which maximizes the number of blue blocks in its past. The block from which the coloring is inherited will be termed its *coloring parent*. In addition to the inherited coloring, each block colors (according to a given coloring rule) all blocks in its past that weren't colored by its coloring parent.

The inheritance means that for each block, the coloring (and ordering, given the right algorithm) of blocks "behind" its coloring parent is fixed. Thus, saving the changes only, the *diffpast*, provides a solid base for efficient coloring and ordering algorithms, even in cases of so-called "reorganizations", or *reorgs*, events where a previously accepted colouring/ordering for the DAG needs to be changed dramatically as a result of the addition of a new block.

E-mail address: aviv.yaish@mail.huji.ac.il.
Date: June 2018.

When looking at the chain of inheritance, meaning - when starting from a given block *block* and looking at its coloring parent, and the coloring parent of its coloring parent, etc', a *coloring chain* is received, thus *block* can be referred to as the coloring chains *tip*. Because each block on the chain colors past blocks that weren't colored by its parent, this coloring chain induces a coloring on all the past of *block*.

As in Algorithm 1 of [1], in this version the *bluest* of all the tips (the one which maximizes the number of blue blocks in its past) of a given DAG induces the coloring of the entire graph; we'll call this block the coloring tip of the entire DAG. Because the coloring of its past is fixed, it is only left to calculate the coloring and ordering of blocks not in its past, its *antipast*.

By looking at an "imaginary" block whose parents are all blocks with in-degree 0 in the DAG, we get that its coloring parent is the coloring tip of the DAG, and that the blocks that it adds to the coloring are those not colored by its coloring parent - meaning exactly the antipast of the coloring parent. Call this imaginary block the *virtual block* of the DAG, and note that the coloring of its past is exactly the coloring induced on the entire graph.

The above observations give way for an efficient and clean implementation of the PHANTOM protocol, which will now be described.

1.1. **Definitions.** Let's proceed to give formal definitions to some important terms:

Definition 1.1. Block: a graph vertex which can hold data. Each block is recognized by an ID, henceforth referenced to as the block's *global ID*, calculated by taking the hash of the block. Blocks pointed at by outgoing edges of a block are called the block's *parents*.

Definition 1.2. BlockDAG: a directed acyclic graph where each vertex is a block.

Definition 1.3. Genesis block: one can assume every BlockDAG contains at least one block, called the *genesis* block, which is added to the BlockDAG before all other blocks, has a global ID of 0, and is reachable from all other blocks.

Definition 1.4. Local ID: given a DAG G, a linear ordering algorithm ord, and the global ID *block* of a block contained in G, the block's *local ID* according to the linear order is its index in the order, and will be designated by $ord(G, block)$.

Definition 1.5. DAG mapping: given a BlockDAG G and a linear ordering algorithm ord, the mapping according to G, ord is a global ID $\rightarrow$ local ID function such that:

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

Definition 1.6. Coloring: given a BlockDAG G, a selection of a k-cluster (see definition 1 in [1]) in G is called G's coloring. Blocks that are contained in the coloring are called *blue* blocks, whilst all other blocks are called *red* blocks.

Definition 1.7. Coloring rule: given a BlockDAG G and a coloring algorithm $color$, define:

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

Conversely, one can redefine a coloring using a coloring rule:

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

Definition 1.8. Past: given a DAG G and a block with global ID *block* the past graph of *block* in G is the subgraph of G containing all blocks that have a path from *block* to them, and will be designated by: $past(G, block)$.

Definition 1.9. Tip: given a DAG G, a tip of G is a block contained in G such that its in-degree is 0. Given a block with global-id *block* contained in G, denote by $G|_{block}$ the sub-graph of G such that *block* is the sole tip.

Definition 1.10. Virtual block: given a DAG G, its virtual block is a "hypothetical" block, not actually in the DAG, such that its past is the entire graph, meaning that it has an edge to every tip of G. Denote it by $virtual(G)$. Note:

$$past(G, virtual(G)) = G$$

Definition 1.11. Antipast: given a DAG G and a block with global ID *block* the antipast of *block* in G is defined to be:

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

Definition 1.12. Diffpast: given a DAG G, and two blocks $block_1$, $block_2$ such that $block_1 \in past(G, block_2)$, the diffpast of $block_2$ in relation to $block_1$ is defined to be:

$$diffpast(G, block_2, block_1) := past(G, block_2) \setminus past(G, block_1)$$

Definition 1.13. Blue diffpast: given a DAG G, a coloring algorithm *color*, and two blocks $block_1$, $block_2$ such that $block_1 \in past(G, block_2)$, the blue diffpast of $block_1$ according to G, *color* is defined to be:

$$blueDiffpast(G, color, block_2, block_1) := diffpast(G, block_2, block_1) \cap color(G)$$

Usually, $block_1$ will be $block_2$'s coloring parent, and *coloring* will be the coloring induced by $block_2$'s coloring chain, in such cases the following shorthand will be used:

$$blueDiffpast(G, block_2)$$

Definition 1.14. Red diffpast: likewise, we define:

$$redDiffpast(G, color, block_2, block_1) := diffpast(G, block_2, block_1) \setminus blueDiffpast(G, color, block_2, block_1)$$

Definition 1.15. Coloring chain: given a DAG G and a block in it *block*, the coloring chain whose tip is *block* is the sequence $block_n = block$, $block_{n-1}$, ..., $block_0 = genesis$, such that:

$$\forall i \in [n-1] : coloringParent(block_{i+1}) = block_i$$

For brevity, given two blocks b, b' such that b is a coloring ancestor of b', the part of the chain stretching from b' to b can also be written as: $b' \to b$, and the entire coloring chain that starts with b' as: $coloringChain(b')$

Definition 1.16. Main coloring chain: given a DAG G, its main coloring chain is the one whose tip maximizes the number of blue blocks in its past. Denote it by $coloringChain(G)$

Definition 1.17. Chain difference: given two chains $b' \to b$, $c' \to c$, the difference between them is defined to be:

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

Remark. In most cases G is the entire DAG. Thus, in such cases G will be omitted from the parameters as a shorthand.

1.2. **Algorithms.**

1.2.1. *The GreedyPHANTOM data-structure.* The data structure for a GreedyPHANTOM DAG G saves the following properties for each block with global ID *block*:

- Outgoing edges.
- Coloring parent, the block from which this block inherits its coloring.
- Height of the block.
- Total number of blue blocks in the coloring it induces on $G|_{block}$.
- Blue and red diffpasts of the block, each saved as a mapping that includes the index of each block in the diffpasts according to the ordering *block* induces on $G|_{block}$.
- The index of itself in the ordering it induces on $G|_{block}$.

In addition, the blue and red diffpasts are saved for G's virtual block, too, with a slight difference: the diffpast of the virtual block, although being kept up-to-date (see algorithm 5), isn't colored and ordered according to the coloring and ordering induced by the virtual's coloring chain with each addition of a block, as it is not relevant at all for the addition of new blocks, and only necessary when a query regarding the order of one of the blocks in the diffpast of the virtual block is received. Thus, the coloring and ordering of the virtual's diffpast are calculated when such a query is received, if needed.

Algorithms 1 - 7 detail the handling of the data structure.

Remark. Keeping the coloring and ordering of the diffpasts of all blocks is wildly inefficient, memory-wise. But, given the greedy nature of the protocol and its convergence of coloring for blocks "deep" enough in the DAG means that keeping all this data for all blocks is unnecessary - after a block on the main coloring chain is accepted, say *block*, then all its past is accepted, too. Thus, the blue and red diffpasts for all blocks in its past (except of those on the coloring chain) can be safely erased, as it is unused as long as *block* remains a part of the main coloring chain.

In case the pruned data is needed again, it can be recalculated using the same algorithms that calculated it in the first place, repopulating the needed data starting from the nearest main coloring chain block and diffusing outwards.

Thus, after setting an acceptance threshold, following each lengthening of the main coloring chain by one block, the DAG's data-structure can be pruned for all blocks in the (red and blue) diffpast of *block*. This iterative pruning ensures the data structure's size is kept to a minimum.

1.2.2. *Coloring algorithm.* Algorithm 1 of [1] can be redefined recursively in terms of diffpast, and generalized to use any arbitrary coloring rule instead of an anticone based one, as shown in algorithms 9-11.

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

$block \leftarrow \emptyset$
$parents(block) \leftarrow parents$
$data(block) \leftarrow data$
return $block$

Algorithm 2: CREATE-DAG($coloringRule$)

Input : $coloringRule$ - the coloring rule for the DAG
Output: G - a PHANTOM block-DAG

$V \leftarrow \emptyset$
$E \leftarrow \emptyset$
$G \leftarrow (V, E)$
$rule(G) \leftarrow coloringRule$
$genesis(G) \leftarrow \emptyset$
$virtual(G) \leftarrow$ CREATE-BLOCK($\emptyset, \emptyset$)
$coloringChain(G) \leftarrow \emptyset$
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

return $\arg\max\{blueNumber(parent) : parent \in parents(block)\}$
`/* 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

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

To receive algorithm 1 of [1] from algorithm 11 presented here, simply create a new PHANTOM block-DAG using algorithm 2 and the following rule as an input:

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

Then, for the k value specified, calling algorithm 11 on a graph G which was constructed using algorithm 4 will produce the same output as calling algorithm 1 of [1] on G.

In order to add blocks efficiently to a PHANTOM block-DAG, its coloring rule needs to be implemented efficiently. But, it is hard to compute the blue anticone size of an arbitrary block in accordance with an arbitrary coloring, meaning

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

for a block b, even if it is "reasonably" close to the leaves of G.

In lieu of an efficient method and data-structures to retrieve the above, rule 1 wasn't used in the implementation presented here. Instead, the following rule was:

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

Like rule 1, rule 2 "punishes" withheld blocks, whereas the measure here is the number of blue blocks that are referenced by blocks on the main chain of G but not on the coloring chain whose tip is b. Taken together with the greedy selection rule for the main coloring chain, this incentivizes miners to reference all currently known leaves as the parents of a newly mined block, and to publish it as soon as possible.

This rule is efficiently computed using algorithm 9: given a k-chain such that the block of lowest height contained in it is of height h, coloring a given block takes at most $O(height(b) - h)$.

It is possible to think of even more efficiently-computable rules, for example:

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

Which takes on average $O(k)$.

Note that when run on a given block *block*, algorithm 10 performs the following operations once:

- Calculation of $antipast(coloringParent(block))$. The antipast of *block* is derived by taking the antipast of the virtual block, and adding/removing the diffpasts of the blocks on the path from the virtual block to *block* while walking only on the virtual's and *block*'s coloring chain, as shown in 7. The computation of the antipast like so: first, the highest block which is both in $coloringParent(block)$'s coloring chain and the main coloring chain of G is found; this takes on average:

$$O(|(coloringChain(block) - coloringChain(G)) \cap coloringChain(block)|)$$

  Then, the diffpast of every block on the path

$$coloringParent(virtual(G)) \rightarrow intersection$$

is lazily [1] added to *antipast*, and the diffpast of every block on the path

$$coloringParent(block) \rightarrow intersection$$

is lazily removed from *antipast*. All in all, we get an average run-time complexity of

$$O(|coloringChain(block) - coloringChain(G)|)$$

- Calculation of the k-chain for *block*. Except for the genesis, each block adds at least one blue block to the coloring of its coloring parent - the coloring parent itself; as such, the length of the k-chain is at most k. Thus, calculating the k-chain (which is always pre-computed in the actual code) takes $O(k)$ on average.

And the following operations on every block *curBlock* as part of a BFS on $diffpast(block)$:

- Containment check in $antipast(coloringParent(block))$. The check takes the same time as the calculation of the antipast, as the LazySet simply traverses each set that's been added to it once according to the order of addition/removal [2].
- Activation of the coloring rule on *block*'s k-chain and *curBlock*. Denote this by $O(|rule|)$.

Assume $diffpast(block) = (V, E)$. So, in total, algorithm 10 takes on average

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

Note that, when no foul-play is involved, $*$ should be relatively small owing to the mentioned-above incentives of honest miners.

Remark. As the PHANTOM block-DAG data structure remembers the blue diffpast for every block including the virtual block, the coloring of the entire graph according to the greedy selection rule is given by chaining together the diffpast colorings of every block along the main coloring chain, starting with the virtual and finishing with the genesis, as computed in algorithm 11.

In the actual code, a ChainMap is kept constantly updated with references to the diffpast colorings of the main coloring chain. Thus, although reorgs are efficient (taking $O(|\text{difference of chains}|)$ on average), inclusion checks take $O(\text{length of main coloring chain})$ on average.

The coloring of the entire graph and of the diffpast of the coloring tip only matter for queries regarding block order, and take no part at all in the addition of new blocks; so, this is acceptable. But, if a speedup is required, one can can save the coloring set of the past of all main chain blocks that were already accepted, as this set is unlikely to be changed in the future, in a similar fashion to the data structure trimming mentioned in remark 1.2.1.

[1] See appendix 4.
[2] See appendix 4.

1.2.3. *Ordering algorithm.* Like the coloring algorithm, the ordering algorithm proposed lets each block inherit the ordering of its coloring parent, and expands it to its diffpast. By looking at the coloring parent first, this algorithm induces an order that is unchangeable for past blocks and thus can be saved together with the coloring of the diff-past of each block.

When adding a block using algorithm 4, the ordering algorithm is run after algorithm 10 was performed, and as such the red and blue diffpasts of the block are already computed. Algorithm 14 simply goes over the diffpast, assigning an index to each block, according to the following recursive rule: given a block b and a colored diffpast $diffpast$, $coloringParent(b)$ has the lowest possible order index, the following indices are given to $color(diffpast) \cap parents(b)$ with the lower indices being given to blocks with lower global IDs, and the rest given to $parents(b) \setminus color(diffpast)$, again with lower indices given to blocks with lower global IDs, the stopping condition being arriving at an already ordered block.

Assuming that *block*'s diffpast is being ordered, we know that $coloringParents(block)$'s diffpast has been ordered before, thus we also know $coloringParents(block)$'s self index in the ordering it induces on the graph. This index will be given to the first diffpast block according to *block*'s order, which is of course $coloringParent(block)$ itself, as all of $coloringParent(block)$'s parents have been ordered before, when it was added to the graph.

This recursive rule performs a BFS traversal of every block, performing

$$O(|parents(block)| \log |parents(block)|)$$

operations for every block. Assume $diffpast(block) = (V, E)$; in total the algorithm has an average run time complexity of

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

Note that the sorting is required only to make sure all miners that know of the same blocks traverse them in the same order and thus have their order in sync. But, if the blocks data structure keeps the parents set ordered, and the global ID is computed on the entire data structure, including the parents set, then algorithm 13, even without using SORT, will produce the same order every time, eschewing the need for sorting, and giving algorithm 14 a total average complexity of $O(|V|+|E|)$.

Remark. As in remark 1.2.2, the order of the entire block-DAG is actually saved in a ChainMap, giving a query on the local ID of a block the same efficiency, problem, and solution as a query on the color of a block in the graph's main coloring.

1.2.4. *Time complexity.* Algorithm 5. Given that a new block b is added, if it is not the new coloring chain tip of the graph then the run-time is $O(|parents(b)|)$ on average, accounting for line 2.

If it is the new tip, then the coloring chain of the graph is updated. Denote: $b \to b' := coloringChain(b) - coloringChain(b')$, if the previous coloring block is b', this takes on average

$$O(\sum_{c \in b \to b'} |diffpast(c)|)$$

Because of the calculation of $antipast(b)$. Note that assuming no foul-play, this is supposed to be small, and thus - efficient.

Note that $parents(b)$ are included in $diffpast(b)$.

Algorithm 4. Given that a new block b is added to G, that b' is the previous coloring tip of G, assuming $diffpast(b) = (V, E)$ according to previous calculations lines 1-3,7 take $O(|parents(b)|)$, lines 4-7 take

$$O(|E| + |V| \cdot k \cdot |b \to b'| \cdot |rule|)$$

And line 8 was shown previously to take $O(|parents(b)|)$ if b is not the new coloring tip of the graph, and

$$O(\sum_{c \in b \to b'} |diffpast(c)|)$$

If it is. So, in the worst case the run time is:

$$O(\sum_{c \in b \to b'} |diffpast(c)| + |V| \cdot k \cdot |b \to b'| \cdot |rule|)$$

2. Simulation framework

2.1. **Simulation framework.** To test out the PHANTOM implementation, a simulation framework was built, simulating miner and network behavior, and allowing fine-grained control of all involved parameters.

When the simulation is started, the simulated network is populated with miners, according to the given parameters. In addition, it is possible to simulate on-the-fly miner joining and leaving the network.

The simulated miners receive an empty block-DAG as a parameter, assuming it implements a DAG abstract class that contains a minimal interface for miner-DAG interactions. Thus, any block-DAG that is implemented according to this interface can be simulated, too.

The simulation assigns a miner to mine a block in a Poisson process with a given rate, where the choice of the miner is done according to the hash-rate distribution of the entire network. As the simulation doesn't include transactions, the miner simply generates a block with random data and parents chosen according to the DAG used, and then proceeds to transmit the block to all its peers. The network assigns random peers for every miner, and random delay, distributed according to a given parameter, to all peer to peer links.

The peers receive the block and add it to their own DAG if deemed valid, and according to the behavior specified when creating them can either broadcast the block to all their peers (as in [1]) or do nothing.

If a miner hears of a valid block that can't be added because not all of the block's parents are in the miner's DAG, it is added to a to-add queue. Now, according to the behavior specified in the miner's parameters, it can either request the missing parents from all it's neighbors, or simply wait passively until one of it's neighbors broadcasts it unprovoked.

The framework allows the simulation of various attacks. For example, an implementation of a selfish miner is given, where the miner withholds the publishing of mined blocks until instructed to do so by its DAG. In addition, the network behavior of the malicious miners can be controlled too, allowing them to have zero network delay to peers, or to have outgoing links to all miners on the network.

2.2. **Simulation setting.**

2.2.1. *Parameters.* All simulations were run with the following parameters:

- Block creation rate: 1 block/minute.
- 100 honest miners, each with 8 peers.
- Network topology and hash rates were randomized for each single simulation, but keeping the hash ratio of the malicious miner constant.
- Honest miners broadcast every block they receive, but don't automatically fetch blocks requested from them that they don't have.
- Malicious miners have zero network delay and outgoing links to all miners on the network.
- Simulation length was set to allow at least 3 successful attacks by the malicious miners.

2.2.2. *Attack description.* There is a single malicious miner who selfishly mines, trying to perform a double-spending attack by starting in private a "competing" chain in parallel to the attacked block, trying to out-run the honest chain, hopefully changing the topological order of the graph such that the first malicious block comes before the attacked block. If that is the case when the honest miners accept the attacked block, and if the malicious miner has mined enough blocks to allow the first malicious block to be accepted after revealing the competing chain in its entirety, the competing chain is broadcast to all miners on the network.

For every new malicious block *new*, the malicious miner chooses the previous malicious block *prev* and the honest blocks closest to the tip such that their blue past is smaller than *prev*'s, thus we assure that *prev* will be the coloring parent of the new malicious block. See algorithm 16 for more details. Denote $antipast(prev) = (V, E)$ and note the algorithm's run time is $O(|V| + |E|)$ on average.

According to the given parent choice rule, if a malicious block is the coloring tip of the DAG then the coloring chain will pass through the first malicious block, which is parallel to the attacked block. According to the ordering algorithm, it means that the malicious block will have a lower local ID than the attacked block; thus any conflicting pair of transactions in the two blocks will be resolved according to the malicious block. By publishing the malicious chain only after the honest block is accepted by the honest miners and the malicious block is accepted by the malicious miner, a double spending attack will be successfully performed.

The malicious miner aborts and restarts the attack if it is deemed that the honest chain passes his malicious chain by more than his own hash ratio times the block confirmation depth.

3. Results & conclusions

Various values for k, block confirmation depth and malicious hash ratio (relative to the entire network) were tested, where each parameter combination was simulated at least until achieving a 5% error margin and uncertainty.

As seen in figures 1 - 3, correct parametrization of k and acceptance depth is important to ensure the security of the protocol, which, as assumed, increases with lower k and higher acceptance depth values.

Note that although for $k > 0$ a large confirmation depth is needed (compared to Blockchain), the PHANTOM protocol allows this depth to be met quickly by setting the block creation rate to be higher. There are various propositions (as in [2]) to

use the block-DAG nature of the protocol to allow high transaction throughput while maintaining a low block size, allowing rapid transmission between miners.

To conclude, given the right parameters, PHANTOM is secure and can be efficiently implemented.

4. Future work

Transaction simulation. Incorporating transactions into the simulation, simulating miner transaction selection, and measuring transaction and block throughput of the algorithms can be of considerable interest for a real-world implementation of the protocol as part of a cryptocurrency. For example, it is interesting to measure rule 2 vs rule 3 with regards to throughput and transaction confirmation times.

Designing better attacks. The attack presented here is not proven to be the optimal double spending attack. One can think of various propositions that might work better, for example changing the parent selection scheme (e.g. instead of selecting only less "blue" parents, it is also possible to select parents with coloring chains that don't pass through the attacked block), or even public attacks.

Internet simulations. The simulation framework as presented here doesn't use the "real-world" internet network for inter-miner communications, rather it simulates delays itself. But, the Miner class uses the networking functionality as a black-box, allowing easy implementation of a Network module that actually uses the internet, allowing simulation in real-world conditions.

Appendices

Brute-force PHANTOM. In addition to the efficient greedy implementation of PHANTOM presented here, also implemented in the code is the "original" version of PHANTOM, which uses rule 1 together with a coloring of the graph that achieves the maximal possible number of blue blocks in the graph. But, as this algorithm is in NP, the implementation was done using brute-force.

LazySet. As part of the need to calculate a block's antipast efficiently, the LazySet data structure was developed. Like a regular set, a user can check if a certain item is contained in the set, remove and add items, and perform set unions, differences, intersections, and more.

But, while a regular set s performs a union/difference with another set s' such that $|s'| = n$ in $O(n)$, a LazySet does so in $O(1)$, trading off the containment check run-time complexity, which takes at most O(# of sets involved in the union/difference).

In addition, note that a LazySet doesn't duplicate the items included in any of the sets it is composed of, simply keeping a reference to each set, thus saving memory.

More info can be found here.

Acknowledgment

The author would like to thank Yonatan Sompolinsky and Aviv Zohar for help and guidance on the PHANTOM protocol, and Tamir Segal and Chaim Hoch for company and chit chat.

FIGURE 1. Attack success rate as a function of k, when using rule 2

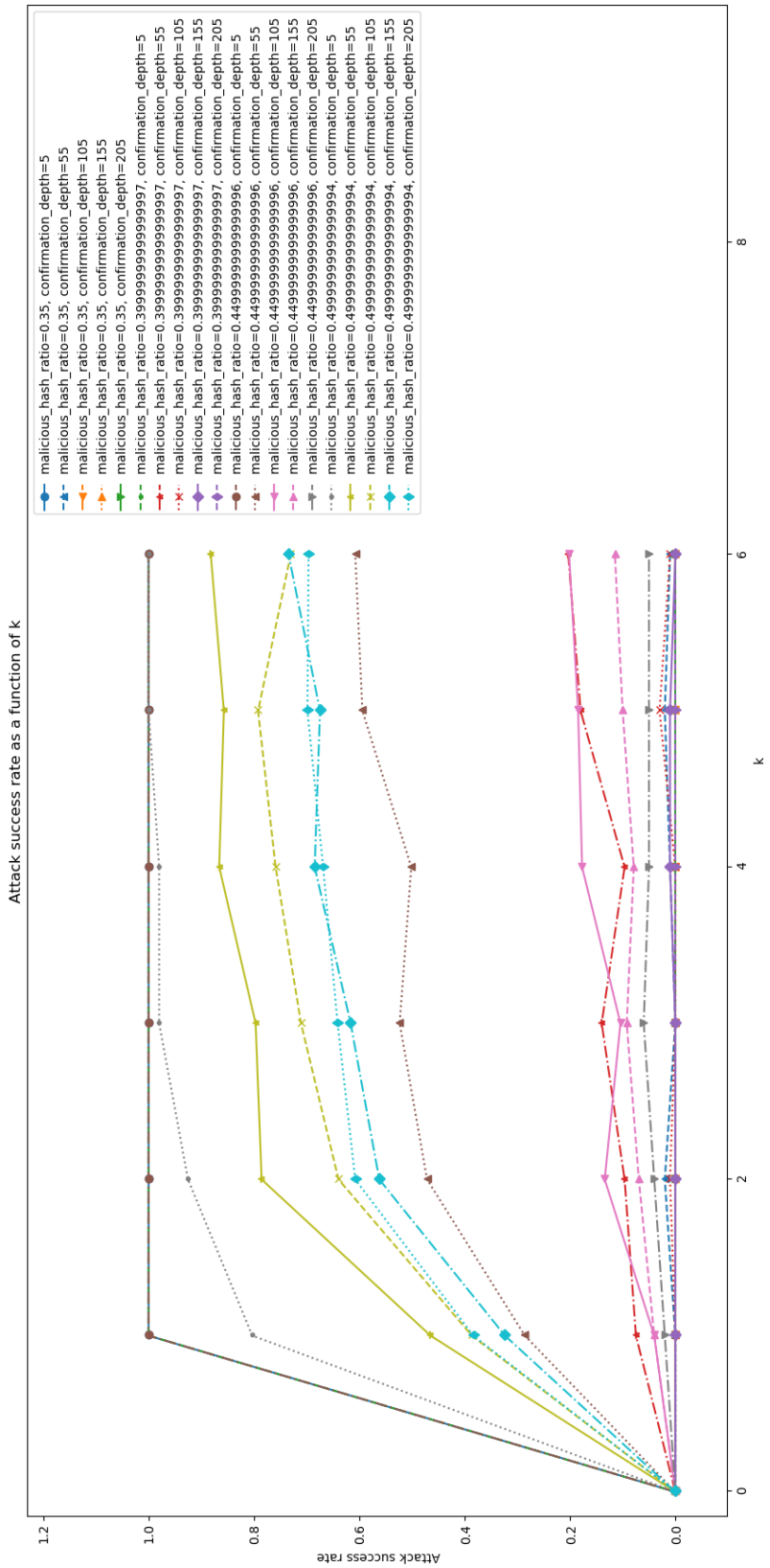


FIGURE 2. Attack success rate as a function of confirmation depth, when using rule 2

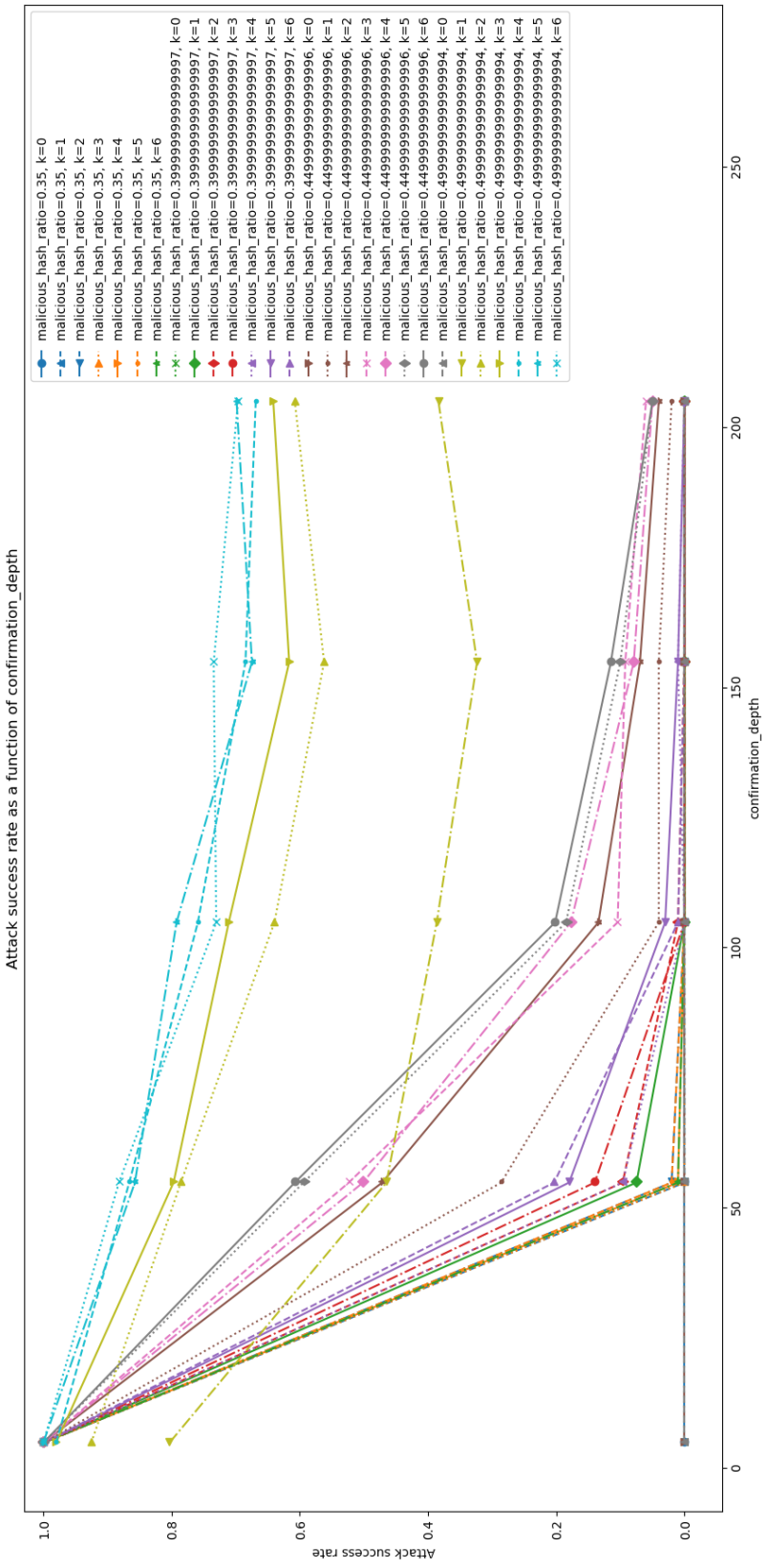
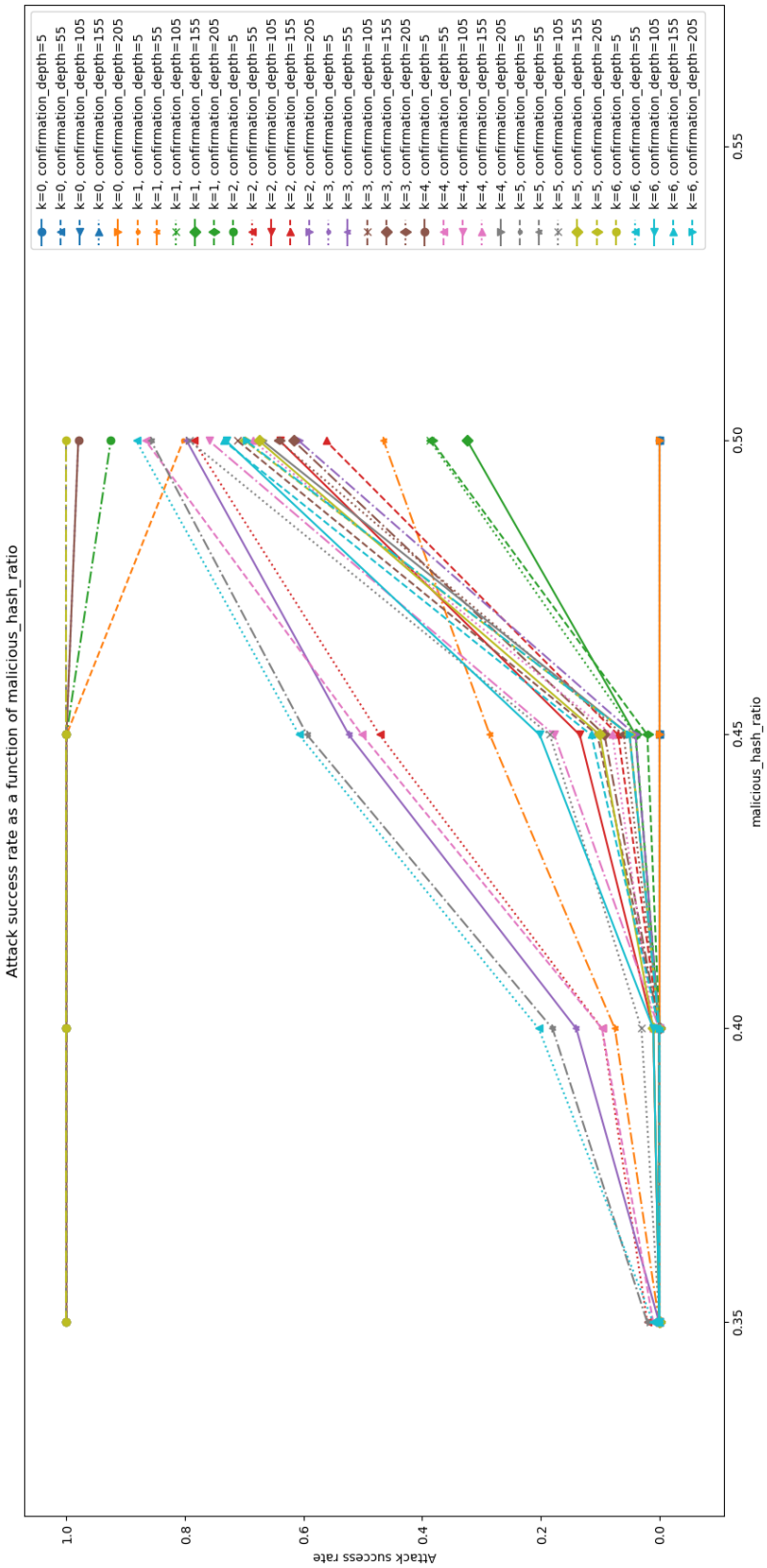


FIGURE 3. Attack success rate as a function of malicious hash ratio, when using rule 2



References

[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

$virtual \leftarrow virtual(G)$
$parents(virtual) \leftarrow (parents(virtual) \cup \{block\}) \setminus parents(block)$
$diffpast(virtual) \leftarrow diffpast(virtual) \cup block$
$order(diffpast(virtual)) \leftarrow \emptyset$
$oldColoringParent \leftarrow coloringParent(virtual)$
$coloringParent(virtual) \leftarrow$ GET-COLORING-PARENT($virtual$)
`/* The actual code uses a flat LazySet for the antipast here, allowing better efficiency: */`
$diffpast(virtual) \leftarrow$ GET-ANTIPAST($colorintParent(virtual)$
$intersection \leftarrow$ GET-MAX-CHAIN-INTERSECTION($coloringParent(virtual)$)
for $curBlock \in oldColoringParent \rightarrow intersection$ **do**
  $coloringChain(G) \leftarrow coloringChain(G) \setminus \{curBlock\}$
end
for $curBlock \in coloringParent(virtual) \rightarrow intersection$ **do**
  $coloringChain(G) \leftarrow coloringChain(G) \cup \{curBlock\}$
end
if $genesis(G) \notin coloringChain(G)$ **then**
  $genesis(G) \leftarrow \arg\min\{height(b) : b \in coloringChain(G)\}$
end
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 $coloringChain(G)$

$intersection \leftarrow block$
for $curBlock \in coloringChain(block)$ **do**
  if $block \in coloringChain(G)$ **then**
    return $curBlock$
  end
end
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

$antipast \leftarrow$ CREATE-LAZYSET()
$intersection \leftarrow$ GET-MAX-CHAIN-INTERSECTION($coloringParent(virtual)$)
for $curBlock \in virtual(G) \rightarrow intersection$ **do**
  $antipast \leftarrow antipast \cup diffpast(curBlock)$
end
for $curBlock \in block \rightarrow intersection$ **do**
  $antipast \leftarrow antipast \setminus diffpast(curBlock)$
end
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 $coloringChain(G)$

$kChain \leftarrow \emptyset$
$depth \leftarrow 0$
for $curBlock \in coloringChain(G)$ **do**
  if $depth > k$ **then**
    return $kChain$
  end
  $kChain \leftarrow kChain \cup curBlock$
  $minHeight(kChain) \leftarrow height(curBlock)$
  $depth \leftarrow depth + |blue(diffpast(curBlock))|$
end
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.                                  */
```

$kChain \leftarrow$ CALC-K-CHAIN(k)
for $curBlock \in coloringChain(block)$ **do**
  if $block \in kChain$ **then**
    return $BLUE$
  end
  if $height(block) < minHeight(kChain)$ **then**

```
    /* Intersection surely occurs only at a lower height, thus
       the block is definitely red                            */
```

    return RED
  end
end

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

Input : G - a block DAG,
block - a block contained in G
Output: the coloring of $diffpast(G, block)$ according to G's coloring rule

$RULE \leftarrow rule(G)$
$blue(diffpast(block)) \leftarrow \emptyset$
$red(diffpast(block)) \leftarrow \emptyset$
$visited \leftarrow \emptyset$
$coloringParentAntipast \leftarrow$ GET-ANTIPAST(coloringParent(block))
$toVisit \leftarrow$ CREATE-QUEUE($parents(block)$)
while $toVisit \neq \emptyset$ **do**
  $curBlock \leftarrow$ POP($toVisit$)
  /* $diffpast(block) = antipast(coloringParent(block)) \cap past(block)$ */
  if $(curBlock \notin visited) \& (curBlock \in coloringParentAntipast)$ **then**
    $visited \leftarrow visited \cup curBlock$
    $toVisit \leftarrow toVisit \cup parents(curBlock)$
    if $RULE(G|_{block}, curBlock) = BLUE$ **then**
      $blue(diffpast(block)) \leftarrow blue(diffpast(block)) \cup \{curBlock\}$
    else
      $red(diffpast(block)) \leftarrow red(diffpast(block)) \cup \{curBlock\}$
    end
  end
end
return $blue(diffpast(block)), red(diffpast(block))$

Algorithm 11: GET-BLUE(G)

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

$coloring \leftarrow$ CREATE-LAZYSET()
for $curBlock \in virtual(G) \rightarrow genesis(G)$ **do**
  $coloring \leftarrow coloring \cup blue(diffpast(curBlock))$
end
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

if $block \notin G$ **then**
  return $-\infty$
end
if $block \in diffpast(virtual(G))$ **then**
  return 0
end
`/* From now on, all blocks are behind G's coloring tip */`
$depth \leftarrow 1$
for $curBlock \in coloringChain(G)$ **do**
  if $block \in red(diffpast(block))$ **then**
    return 0
  end
  if $block \in blue(diffpast(block))$ **then**
    return $depth$
  end
  $depth \leftarrow depth + |blue(diffpast(block))|$
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 $toSort \cap diffpast$

$sorted \leftarrow$ CREATE-LIST()
$remainingBlocks \leftarrow (toSort \setminus \{last\}) \cap diffpast$
`/* SORT is a descending sorting algorithm on the global IDs */`
$blueSorted \leftarrow$ SORT($remainingBlocks \cap color(diffpast)$}
$redSorted \leftarrow$ SORT($remainingBlocks \setminus blueSorted$)}
$sorted \leftarrow redSorted \cup blueSorted$
if $last \neq \emptyset$ **then**
  $sorted \leftarrow sorted \cup \{last\}$
end
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 $diffpast(block)$

if $coloringParent(block) \neq \emptyset$ **then**
  $curIndex \leftarrow selfIndex(coloringParent(block))$
else
  $curIndex \leftarrow 0$
end
`/* The code uses a dictionary as the map */`
$order \leftarrow$ CREATE-MAP()
$toOrder \leftarrow$ CREATE-STACK()
$toOrder \leftarrow toOrder \cup block$
while $toOrder \neq \emptyset$ **do**
  $curBlock \leftarrow$ POP($toOrder$)
  if $curBlock \notin order$ **then**
    $curParents \leftarrow parents(curBlock) \cap diffpast(block)$
    if $curParents \subseteq order$ **then**
      if $curBlock \neq block$ **then**
        $order(curBlock) \leftarrow curIndex$
        $curIndex \leftarrow curIndex + 1$
      end
    else
      $toOrder \leftarrow (toOrder \cup \{curBlock\}) \cup$
       SORT-BLOCKS($diffpast(block)$, $coloringParent(curBlock)$,
       $curParents$)
    end
  end
end
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

if $block \notin G$ **then**
  return ∞
end
$virtual \leftarrow virtual(G)$
if $(block \in diffpast(virtual))$ *and* $(order(diffpast(virtual)) = \emptyset)$ **then**
  if $color(diffpast(virtual)) = \emptyset$ **then**
    $color(diffpast(virtual)) \leftarrow$ CALC-DIFFPAST-COLOR($virtual$)
  end
  $order(diffpast(virtual)) \leftarrow$ CALC-DIFFPAST-ORDER($virtual$)
end
for $curBlock \in virtual \rightarrow genesis(G)$ **do**
  $orderMap \leftarrow order(diffpast(curBlock))$
  if $block \in orderMap$ **then**
    return GET-VALUE($orderMap, block$)
  end
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 ← GET-ANTIPAST(coloringParent(block))
2  maliciousParents ← ∅
3  visited ← ∅
4  toVisit ← CREATE-QUEUE()
5  toVisit ← toVisit ∪ parents(virtual(G))
6  while toVisit ≠ ∅ do
7      curBlock ← POP(toVisit)
8      visited ← visited ∪ {curBlock}
9      if (blueNumber(curBlock) <
        blueNumber(prev))|((blueNumber(curBlock) =
        blueNumber(prev))&(prev <= curBlock)) then
10         maliciousParents ← maliciousParents ∪ {curBlock}
11         ancestors ← CREATE-QUEUE()
12         ancestors ← ancestors ∪ parents(curBlock)
13         while ancestors ≠ ∅ do
14             curAncestor ← POP(ancestors)
15             if curAncestor ∈ parentAntipast then
16                 visited ← visited ∪ {curAncestor}
17                 maliciousParents ← maliciousParents \ {curAncestor}
18                 ancestors ← ancestors ∪ parents(curAncestor)
19             end
20         end
21     else
22         toVisit ← toVisit ∪ parents(curBlock)
23     end
24 end
25 return maliciousParents
```