Efficient Recovery from False State in Distributed Routing Algorithms

Daniel Gyllstrom, Sudarshan Vasudevan, Jim Kurose, Gerome Miklau
Department of Computer Science
University of Massachusetts Amherst
{dpg, svasu, kurose, miklau}@cs.umass.edu

***Abstract*—Malicious and misconfigured nodes can inject incorrect state into a distributed system, which can then be propagated system-wide as a result of normal network operation. Such false state can degrade the performance of a distributed system or render it unusable. For example, in the case of network routing algorithms, false state corresponding to a node incorrectly declaring a cost of 0 to all destinations (maliciously or due to misconfiguration) can quickly spread through the network. This causes other nodes to (incorrectly) route via the misconfigured node, resulting in suboptimal routing and network congestion. We propose three algorithms for efficient recovery in such scenarios, prove the correctness of each of these algorithms, and derive communication complexity bounds for each algorithm. Through simulation, we evaluate our algorithms – in terms of message and time overhead – when applied to removing false state in distance vector routing. Our analysis shows that over topologies where link costs remain fixed and for the same topologies where link costs change, a recovery algorithm based on system-wide checkpoints and a rollback mechanism yields superior performance when using the poison reverse optimization.**

I. Introduction

Malicious and misconfigured nodes can degrade the performance of a distributed system by injecting incorrect state information. Such false state can then be further propagated through the system either directly in its original form or indirectly, e.g., as a result of diffusing computations initially using this false state. In this paper, we consider the problem of removing such false state from a distributed system.

In order to make the false-state-removal problem concrete, we investigate distance vector routing as an instance of this problem. Distance vector forms the basis for many routing algorithms widely used in the Internet (e.g., BGP, a path-vector algorithm) and in multi-hop wireless networks (e.g., AODV, diffusion routing). However, distance vector is vulnerable to compromised nodes that can potentially flood a network with false routing information, resulting in erroneous least cost paths, packet loss, and congestion. Such scenarios have occurred in practice. For example, in 1997 a significant portion of Internet traffic was routed through a single misconfigured router, rendering a large part of the Internet inoperable for several hours [22]. More recently [1], a routing error forced Google to redirect its traffic through Asia, causing congestion that left many Google services unreachable. Distance vector currently has no mechanism to recover from such scenarios. Instead, human operators are left to manually reconfigure routers. It is in this context that we propose and evaluate automated solutions for recovery.

In this paper, we design, develop, and evaluate three different approaches for correctly recovering from the injection of false routing state (e.g., a compromised node incorrectly claiming a distance of 0 to all destinations). Such false state, in turn, may propagate to other routers through the normal execution of distance vector routing, making this a network-wide problem. Recovery is correct if the routing tables in all nodes have converged to a global state in which all nodes have removed each compromised node as a destination, and no node bears a least cost path to any destination that routes through a compromised node.

Specifically, we develop three novel distributed recovery algorithms: 2$^{\text{nd}}$ `best`, `purge`, and `cpr`. 2$^{\text{nd}}$ `best` performs localized state invalidation, followed by network-wide recovery. Nodes directly adjacent to a compromised node locally select alternate paths that avoid the compromised node; the traditional distributed distance vector algorithm is then executed to remove remaining false state using these new distance vectors. The `purge` algorithm performs global false state invalidation by using diffusing computations to invalidate distance vector entries (network-wide) that routed through a compromised node. As in 2$^{\text{nd}}$ `best`, traditional distance vector routing is then used to recompute distance vectors. `cpr` uses local snapshots and a rollback mechanism to implement recovery. Although our solutions are tailored to distance vector routing, we believe they represent approaches that are applicable to other instances of this problem.

We prove the correctness of each algorithm, derive communication complexity bounds for each algorithm, and use simulations to evaluate the efficiency of each algorithm in terms of message overhead and convergence time. Our simulations show that `cpr` using poison reverse outperforms 2$^{\text{nd}}$ `best` and `purge` (with and without poison reverse) – at the cost of checkpoint memory – over topologies with fixed and changing link costs. This is because `cpr` efficiently removes all false state by rolling back to a checkpoint immediately preceding the injection of false routing state. In scenarios where link costs can change, `purge` using poison reverse yields performance close to `cpr` with poison reverse. `purge` makes use of computations subsequent to the injection of false routing state that do not depend on false routing state, while `cpr` must process all valid link cost changes that occurred since false routing state was injected. Finally, our simulations show that poison reverse significantly improves performance for all three algorithms, especially for topologies with changing link costs.

Recovery from false routing state is closely related to the problem of recovering from malicious transactions [4], [17] in distributed databases. Our problem is also similar to that of rollback in optimistic parallel simulation [13]. However, we are unaware of any existing solutions to the problem of recovering from false routing state. A closely related problem to the one considered in this paper is that of discovering misconfigured nodes. In Section II, we discuss existing solutions to this problem. In fact, the output of these algorithms serve as input to the recovery algorithms proposed in this paper.

This paper has eight sections. In Section II we define the problem and state our assumptions. We present our three recovery algorithms in Section III and prove the correctness of each algorithm in Section IV. Then, in Section V, we present a qualitative evaluation of our recovery algorithms and derive communication complexity bounds for each algorithm. Section VI describes our simulation study. We detail related work in Section VII and finally we conclude and comment on directions for future work in Section VIII.

II. Problem Formulation

We consider distance vector routing [5] over arbitrary network topologies. We model a network as an undirected graph, $G = (V, E)$, with a link weight function $w : E \rightarrow \mathbb{N}$. [1] Each node, v, maintains the following state as part of distance vector: a vector of all adjacent nodes ($adj(v)$), a vector of least cost distances to all nodes in G ($\overrightarrow{min_v}$), and a *distance matrix* that contains distances to every node in the network via each adjacent node ($dmatrix_v$).

For simplicity, we present our recovery algorithms in the case of a single compromised node. We describe the necessary extensions to handle multiple compromised nodes in Section III-E.

We assume that the identity of the compromised node is provided by a different algorithm, and thus do not consider this problem in this paper. Examples of such algorithms include [7], [8], [19], [23], [25]. Specifically, we assume that at time t_b, this algorithm is used to notify all neighbors of the compromised node. Let t' be the time the node was compromised.

For each of our algorithms, the goal is for all nodes to recover "correctly": all nodes should remove the compromised nodes as a destination and find new least cost distances that do not use a compromised node. If the network becomes disconnected as a result of removing the compromised node, all nodes need only compute new least cost distances to all other nodes within their connected component. For simplicity, let $\overline{v}$ denote the compromised node, let $\overrightarrow{old}$ refer to $\overrightarrow{min_{\overline{v}}}$ before $\overline{v}$ was compromised, and let $\overrightarrow{bad}$ denote $\overrightarrow{min_{\overline{v}}}$ after $\overline{v}$ has been compromised. Intuitively, $\overrightarrow{old}$ and $\overrightarrow{bad}$ are snapshots of the compromised node's least cost vector taken at two different timesteps: $\overrightarrow{old}$ marks the snapshot taken before $\overline{v}$ was compromised and $\overrightarrow{bad}$ represents a snapshot taken after $\overline{v}$ was compromised.

[1]Recovery is simple with link state routing: each node uses its complete topology map to compute new least cost paths that avoid all compromised nodes. Thus we do not consider link state routing in this paper.

Abbreviation	Meaning
$\overrightarrow{min_i}$	node i's the least cost vector
$dmatrix_i$	node i' distance matrix
DV	Distance Vector
t_b	time the compromised node is detected
t'	time the compromised node was compromised
$\overrightarrow{bad}$	compromised node's least cost vector at and after t
$\overrightarrow{old}$	compromised node's least cost vector at and before t'
$\overline{v}$	compromised node
$adj(v)$	nodes adjacent to v in G'

TABLE I
Table of abbreviations.

Table I summarizes the notation used in this document.

III. Recovery Algorithms

In this section we propose three new recovery algorithms: 2$^{\textbf{nd}}$ `best`, `purge`, and `cpr`. With one exception, the input and output of each algorithm is the same. [2]

Input: Undirected graph, $G = (V, E)$, with weight function $w : E \rightarrow \mathbb{N}$. $\forall v \in V$, $\overrightarrow{min_v}$ and $dmatrix_v$ are computed (using distance vector). Also, each $v \in adj(\overline{v})$ is notified that $\overline{v}$ was compromised.

Output: Undirected graph, $G' = (V', E')$, where $V' = V - \{\overline{v}\}$, $E' = E - \{(\overline{v}, v_i) \mid v_i \in adj(\overline{v})\}$, and link weight function $w : E \rightarrow \mathbb{N}$. $\overrightarrow{min_v}$ and $dmatrix_v$ are computed via the algorithms discussed below $\forall v \in V'$.

First we describe a preprocessing procedure common to all three recovery algorithms. Then we describe each recovery algorithm.

A. Preprocessing

All three recovery algorithms share a common preprocessing procedure. The procedure removes $\overline{v}$ as a destination and finds the node IDs in each connected component. This is implemented using diffusing computations [6] initiated at each $v \in adj(\overline{v})$. A diffusing computation is a distributed algorithm started at a source node which grows by sending queries along a spanning tree, constructed simultaneously as the queries propagate through the network. When the computation reaches the leaves of the spanning tree, replies travel back along the tree towards the source, causing the tree to shrink. The computation eventually terminates when the source receives replies from each of its children in the tree.

In our case, each diffusing computation message contains a vector of node IDs. When a node receives a diffusing computation message, the node adds its ID to the vector and removes $\overline{v}$ as a destination. At the end of the diffusing computation,

[2]Additionally, as input `cpr` requires that each $v \in adj(\overline{v})$ is notified of the time, t', in which $\overline{v}$ was compromised.

each $v \in adj(\overline{v})$ has a vector that includes all nodes in v's connected component. Finally, each $v \in adj(\overline{v})$ broadcasts the vector of node IDs to all nodes in their connected component. In the case where removing $\overline{v}$ partitions the network, each node will only compute shortest paths to nodes in the vector.

Consider the example in Figure 1 where $\overline{v}$ is the compromised node. When i receives the notification that $\overline{v}$ has been compromised, i removes $\overline{v}$ as a destination and then initiates a diffusing computation. i creates a vector and adds its node ID to the vector. i sends a message containing this vector to j and k. Upon receiving i's message, j and k both remove $\overline{v}$ as a destination and add their own ID to the message's vector. Finally, l and d receive a message from j and k, respectively. l and d add their node own ID to the message's vector and remove $\overline{v}$ as a destination. Then, l and d send an ACK message back to j and k, respectively, with the complete list of node IDs. Eventually when i receives the ACKs from j and k, i has a complete list of nodes in its connected component. Finally, i broadcasts the vector of node IDs in its connected component.

B. The 2^{nd} best Algorithm

2^{nd} `best` invalidates state locally and then uses distance vector to implement network-wide recovery. Following the preprocessing described in Section III-A, each neighbor of the compromised node locally invalidates state by selecting the least cost pre-existing alternate path that does not use the compromised node as the first hop. The resulting distance vectors trigger the execution of traditional distance vector to remove the remaining false state. Algorithm 1 in the Appendix gives a complete specification of 2^{nd} `best`.

We trace the execution of 2^{nd} `best` using the example in Figure 1. In Figure 1(b), i uses $\overline{v}$ to reach nodes l and d. j uses i to reach all nodes except l. Notice that when j uses i to reach d, it transitively uses $\overrightarrow{bad}$ (e.g., uses path $j-i-\overline{v}-d$ to d). After the preprocessing completes, i selects a new neighbor to route through to reach l and d by finding its new smallest distance in $dmatrix_i$ to these destinations: i selects the routes via j to l with a cost of 100 and i picks the route via k to reach d with cost of 100. (No changes are required to route to j and k because i uses its direct link to these two nodes). Then, using traditional distance vector i sends $\overrightarrow{min_i}$ to j and k. When j receives $\overrightarrow{min_i}$, j must modify its distance to d because $\overrightarrow{min_i}$ indicates that i's least cost to d is now 100. j's new distance value to d becomes 150, using the path $j-i-k-l$. j then sends a message sharing $\overrightarrow{min_j}$ with its neighbors. From this point, recovery proceeds according by using traditional distance vector.

2^{nd} `best` is simple and makes no synchronization assumptions. However, 2^{nd} `best` is vulnerable to the count-to-∞ problem. Because each node only has local information, the new shortest paths may continue to use $\overline{v}$. For example, if $w(k,d) = 400$ in Figure 1, a count-to-∞ scenario would arise. After notification of $\overline{v}$'s compromise, i would select the route via j to reach d with cost 151 (by consulting $dmatrix_i$), using a path that does not actually exist in G ($i-j-i-\overline{v}-d$), since j has removed $\overline{v}$ as a neighbor. When i sends $\overrightarrow{min_i}$ to j, j selects the route via i to d with cost 201. Again, the path $j-i-j-i-\overline{v}-d$ does not exist. In the next iteration, i picks the route via j having a cost of 251. This process continues until each node finds their correct least cost to d. We will see in our simulation study that the count-to-∞ problem can incur significant message and time costs.

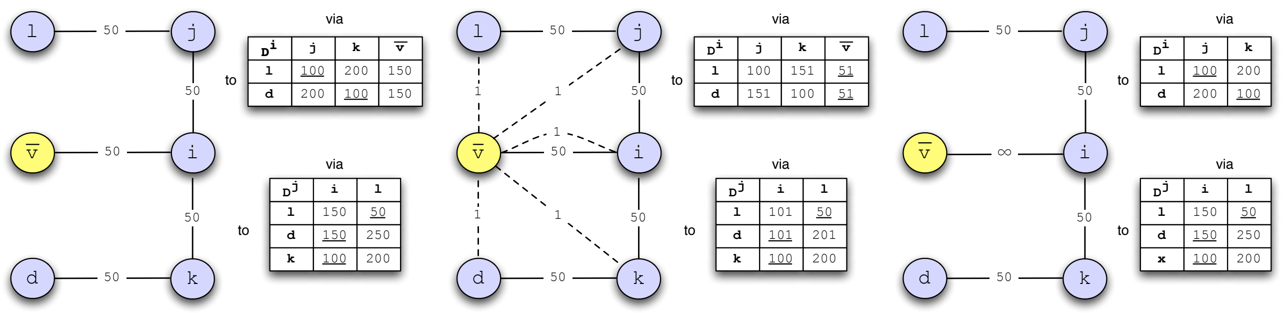
C. The purge Algorithm

`purge` globally invalidates all false state using a diffusing computation and then uses distance vector to compute new distance values that avoid all invalidated paths. Recall that diffusing computations preserve the decentralized nature of distance vector. The diffusing computation is initiated at the neighbors of $\overline{v}$ because only these nodes are aware if $\overline{v}$ is used an intermediary node. The diffusing computations spread from $\overline{v}$'s neighbors to the network edge, invalidating false state at each node along the way. Then ACKs travel back from the network edge to the neighbors of $\overline{v}$, indicating that the diffusing computation is complete. See Algorithm 2 and 3 in the Appendix for a complete specification of this diffusing computation.

Next, `purge` uses distance vector to recompute least cost paths invalidated by the diffusing computations. In order to initiate the distance vector computation, each node is required to send a message after diffusing computations complete, even if no new least cost is found. Without this step, distance vector may not correctly compute new least cost paths invalidated by the diffusing computations. For example, consider the following the scenario when the diffusing computations complete: a node i and all of i's neighbors have least cost of ∞ to destination node a. Without forcing i and its neighbors to send a message after the diffusing computations complete, neither i nor i's neighbors may never update their least cost to a because they may never receive a non-∞ cost to a.

In Figure 1, the diffusing computation executes as follows. First, i sets its distance to l and d to ∞ (thereby invalidating i's path to l and d) because i uses $\overline{v}$ to route these nodes. Then, i sends a message to j and k containing l and d as invalidated destinations. When j receives i's message, j checks if it routes via i to reach l or d. Because j uses i to reach d, j sets its distance estimate to d to ∞. j does not modify its least cost to l because j does not route via i to reach l. Next, j sends a message that includes d as an invalidated destination. l performs the same steps as j. After this point, the diffusing computation ACKs travel back towards i. When i receives an ACK, the diffusing computation is complete. At this point, i needs to compute new least costs to node l and d because i's distance estimates to these destinations are ∞. i uses $dmatrix_i$ to select its new route to l (which is via j) and uses $dmatrix_i$ to find i's new route to d (which is via k). Both new paths have cost 100. Finally, i sends $\overrightarrow{min_i}$ to its neighbors, triggering the execution of distance vector to recompute the remaining distance vectors.

Note that a consequence of the diffusing computation is that not only is all $\overrightarrow{bad}$ state deleted, but all $\overrightarrow{old}$ state as well. Consider the case when $\overline{v}$ is detected before node i receives



D^i	j	k	$\overline{v}$
l	$\underline{100}$	200	150
d	200	$\underline{100}$	150

D^j	i	l
l	150	$\underline{50}$
d	$\underline{150}$	250
k	$\underline{100}$	200

(a) Before $\overline{v}$ is compromised.

D^i	j	k	$\overline{v}$
l	100	151	$\underline{51}$
d	151	100	$\underline{51}$

D^j	i	l
l	101	$\underline{50}$
d	$\underline{101}$	201
k	$\underline{100}$	200

(b) After $\overline{v}$ is compromised.

D^i	j	k
l	$\underline{100}$	200
d	200	$\underline{100}$

D^j	i	l
l	150	$\underline{50}$
d	$\underline{150}$	250
x	$\underline{100}$	200

(c) After recovery.

Fig. 1. Three snapshots of a graph, G, where $\overline{v}$ is the compromised node: (a) G before $\overline{v}$ is compromised, (b) G after $\overrightarrow{bad}$ has finished propagating but before recovery has started, and (c) G after recovery. The dashed lines in (b) mark false paths used by $\overrightarrow{bad}$. Portions of $dmatrix_i$ and $dmatrix_j$ are displayed to the right of each sub-figure. The least cost values are underlined.

$\overrightarrow{bad}$. It is possible that i uses $\overrightarrow{old}$ to reach a destination, d. In this case, the diffusing computation will set i's distance to d to ∞.

An advantage of `purge` is that it makes no synchronization assumptions. Also, the diffusing computations ensure that the count-to-∞ problem does not occur by removing false state from the entire network. However, globally invalidating false state can be wasteful if valid alternate paths are locally available.

D. *The cpr Algorithm*

`cpr`[3] is our third and final recovery algorithm. Unlike `2nd best` and `purge`, `cpr` only requires that clocks across different nodes be loosely synchronized i.e. the maximum clock offset between any two nodes is assumed to be δ. For ease of explanation, we describe `cpr` as if the clocks at different nodes are perfectly synchronized. Extensions to handle loosely synchronized clocks should be clear. Accordingly, we assume that all neighbors of $\overline{v}$, are notified of the time, t', at which $\overline{v}$ was compromised.

For each node, $i \in G$, `cpr` adds a time dimension to $\overrightarrow{min_i}$ and $dmatrix_i$, which `cpr` then uses to locally archive a complete history of values. Once the compromised node is discovered, the archive allows the system to rollback to a system snapshot from a time before $\overline{v}$ was compromised. From this point, `cpr` needs to remove $\overline{v}$ and $\overrightarrow{old}$ and update stale distance values resulting from link cost changes. We describe each algorithm step in detail.

Step 1: Create a $\overrightarrow{min}$ and $dmatrix$ archive. We define a *snapshot* of a data structure to be a copy of all current distance values along with a timestamp. [4] The timestamp marks the time at which that set of distance values start being used. $\overrightarrow{min}$ and $dmatrix$ are the only data structures that need to be archived. This approach is similar to ones used in temporal databases [14], [18].

[3]The name is an abbreviation for **C**heck**P**oint and **R**ollback.

[4]In practice, we only archive distance values that have changed. Thus each distance value is associated with its own timestamp.

Our distributed archive algorithm is quite simple. Each node has a choice of archiving at a given frequency (e.g., every m timesteps) or after some number of distance value changes (e.g., each time a distance value changes). Each node must choose the same option, which is specified as an input parameter to `cpr`. A node archives independently of all other nodes. A side effect of independent archiving, is that even with perfectly synchronized clocks, the union of all snapshots may not constitute a globally consistent snapshot. For example, a link cost change event may only have propagated through part of the network, in which case the snapshot for some nodes will reflect this link cost change (i.e., among nodes that have learned of the event) while for other nodes no local snapshot will reflect the occurrence of this event. We will see that a globally consistent snapshot is not required for correctness.

Step 2: Rolling back to a valid snapshot. Rollback is implemented using diffusing computations. Neighbors of the compromised node independently select a snapshot to roll back to, such that the snapshot is the most recent one taken before t'. Each such node, i, rolls back to this snapshot by restoring the $\overrightarrow{min_i}$ and $dmatrix_i$ values from the snapshot. Then, i initiates a diffusing computation to inform all other nodes to do the same. If a node has already rolled back and receives an additional rollback message, it is ignored. (Note that this rollback algorithm ensures that no reinstated distance value uses $\overrightarrow{bad}$ because every node rolls back to a snapshot with a timestamp less that t'.) Algorithm 4 in the Appendix gives the pseudo-code for the rollback algorithm.

Step 3: Steps after rollback. After Step 2 completes, the algorithm in Section III-A is executed. There are two issues to address. First, some nodes may be using $\overrightarrow{old}$. Second, some nodes may have stale state as a result of link cost changes that occurred during $[t', t_b]$ and consequently are not reflected in the snapshot. To resolve these issues, each neighbor, i, of $\overline{v}$, sets its distance to $\overline{v}$ to ∞ and then selects new least cost values that avoid the compromised node, triggering the execution of distance vector to update the remaining distance vectors. That is, for any destination, d, where i routes via $\overline{v}$ to reach d, i uses $dmatrix_i$ to find a new least cost to d. If a new least costs value is used, i sends a distance vector message to its neighbors. Otherwise, i sends no message. Messages sent trigger the execution of distance vector.

During the execution of distance vector, each node uses the most recent link weights of its adjacent links. Thus, if the same link changes cost multiple times during $[t', t_b]$, we ignore all changes but the most recent one. Algorithm 5 specifies Step 3 of `cpr`.

In the example from Figure 1, the global state after rolling back is nearly the same as the snapshot depicted in Figure 1(c): the only difference between the actual system state and that depicted in Figure 1(c) is that in the former $(i,\overline{v}) = 50$ rather than ∞. Step 3 in `cpr` makes this change. Because no nodes use $\overrightarrow{old}$, no other changes take place.

Rather than using an iterative process to remove false state (like in 2nd `best` and `purge`), `cpr` does so in one diffusing computation. However, `cpr` incurs storage overhead resulting from periodic snapshots of $\overrightarrow{min}$ and $dmatrix$. Also, after rolling back, stale state may exist if link cost changes occur during $[t', t_b]$. This can be expensive to update. Finally, unlike `purge` and 2nd `best`, `cpr` requires loosely synchronized clocks because without a bound on the clock offset, nodes may rollback to highly inconsistent local snapshots. Although correct, this would severely degrade `cpr` performance.

E. Multiple Compromised Nodes

Here we detail the necessary changes to each of our recovery algorithms when multiple nodes are compromised. Since we make the same changes to all three algorithms, we do not refer to a specific algorithm in this section. Let $\overline{V}$ refer to the set of nodes compromised at time t'.

In the case where multiple nodes are simultaneously compromised, each recovery algorithm is modified such that for each $\overline{v} \in \overline{V}$, all $adj(\overline{v})$ are notified that $\overline{v}$ was compromised. From this point, the changes to each algorithm are straightforward. For example, the diffusing computations described in Section III-A are initiated at the neighbor nodes of each node in $\overline{V}$. [5]

More changes are required to handle the case where an additional node is compromised during the execution of a recovery algorithm. Specifically, when another node is compromised, $\overline{v}_2$, we make the following change to the distance vector computation of each recovery algorithm. [6] If a node, i, receives a distance vector message which includes a distance value to destination $\overline{v}_2$, then i ignores said distance value and processes the remaining distance values (if any exist) to all other destinations (e.g., where $d \neq \overline{v}_2$) normally. If the message contains no distance information for any other destination $d \neq \overline{v}_2$, then i ignores the message. Because $\overline{v}_2$'s compromise triggers a diffusing computation to remove $\overline{v}_2$ as a destination, each node eventually learns the identity of $\overline{v}_2$, thereby allowing the node execute the specified changes to distance vector.

[5]For `cpr`, t' is set to the time the first node is compromised.

[6]Recall that each of our recovery algorithms use distance vector to complete their computation.

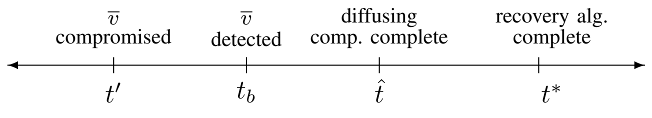


Fig. 2. Time line with important timesteps labeled.

Without this change it is possible that the recovery algorithm will not terminate. Consider the case of two compromised nodes, $\overline{v}_1$ and $\overline{v}_2$, where $\overline{v}_2$ is compromised during the recovery triggered by $\overline{v}_1$'s compromise. In this case, two executions of the recovery algorithm are triggered: one when $\overline{v}_1$ is compromised and the other when $\overline{v}_2$ is compromised. Recall that all three recovery algorithms set all link costs to $\overline{v}_1$ to ∞ (e.g., $(v_i, \overline{v}_1) = \infty, \forall v_i \in adj(\overline{v}_1)$). If the first distance vector execution triggered by $\overline{v}_1$'s compromise is not modified to terminate least cost computations to $\overline{v}_2$, the distance vector step of the recovery algorithm would never complete because the least cost to $\overline{v}_2$ is ∞.

IV. Correctness of Recovery Algorithms

Our correctness proofs consider the general case where multiple nodes are compromised. We use the following notation in our proofs:

- We refer to the set of compromised nodes as $\overline{V}$.
- t_b marks the time at outside algorithm detects that all $\overline{V}$ are compromised.
- t' refers to the time the first $\overline{v} \in \overline{V}$ is compromised.
- t^* marks the time when the recovery algorithm (e.g., 2nd `best`, `purge`, or `cpr`), which started executing at time t, completes.
- We use the definition of G described in Section III.
- We redefine G' as follows. $G' = (V', E')$, where $V' = V - \overline{V}$, $E' = E - \{(\overline{v}, v_i) \mid \overline{v} \in \overline{V} \wedge v_i \in adj(\overline{v})\}$.

All important timesteps are shown in Figure 2.

We make the following assumptions in our proofs. All the initial $dmatrix$ values are non-negative. Furthermore, all $\overrightarrow{min}$ values periodically exchanged between neighboring nodes are non-negative. All $v \in V$ know their adjacent link costs. All link weights in G (and therefore G' as well) are non-negative and do not change. G is finite and connected. Finally, we assume reliable communication.

Definition 1. *An algorithm is correct if the following two conditions are satisfied. One, $\forall v \in V'$, v has the least cost to all destinations $v' \in V'$. Two, the least cost is computed in finite time.*

Theorem 1. *Distance vector is correct.*

Proof: Bertsekas and Gallager [5] prove correctness for distributed Bellman-Ford for arbitrary non-negative $dmatrix$ values. Their distributed Bellman-Ford algorithm is the same as the distance vector algorithm used in this paper. ∎

Corollary 2. 2^{nd} `best` *is correct when a single node is compromised.*

Proof: As per the preprocessing step, each $v \in adj(\overline{v})$ initiates a diffusing computation to remove $\overline{v}$ as a destination. For each diffusing computation, all nodes are guaranteed to receive a diffusing computation (by our reliable communication and finite graph assumptions). Further, each diffusing computation terminates in finite time. Thus, we conclude that each $v \in V'$ removes $\overline{v}$ as a destination in finite time.

After the diffusing computations to remove $\overline{v}$ as a destination complete, each $v \in adj(\overline{v})$ uses distance vector to determine new least cost paths to all nodes in their connected component. Because all $dmatrix_v$ are non-negative for all $v \in V'$, by Theorem 1 we conclude 2^{nd} `best` is correct if no additional node(s) are compromised during $[t', t^*]$. ∎

Corollary 3. 2^{nd} `best` *is correct when multiple nodes are compromised.*

Proof: If multiple nodes, $\overline{V}$, are simultaneously compromised the proof is the same as that for Corollary 2, substituting $\overline{V}$ for $\overline{v}$.

Next, we prove 2^{nd} `best` is correct in the case where a set of nodes, $\overline{V}_2$, are compromised concurrent with a running execution of 2^{nd} `best` (e.g., during $[t', t^*]$), triggered by the compromise of $\overline{V}$. First we show that any least cost computation (e.g., one triggered by $\overline{V}$'s compromise) to any $v \in \overline{V}_2$ is eventually terminated. We have already proved that the diffusing computations to remove each $v \in \overline{V}_2$ as a destination complete in finite time. Let t_d mark the time these diffusing computations complete. For all $t \geq t_d$, any running least cost computation to a destination $v \in \overline{V}_2$ is terminated by the actions specified in Section III-E. Therefore, the only remaining least cost computations are to all $v \in V'$, where $V' = V - \left(\overline{V} \cup \overline{V}_2\right)$. Because all $dmatrix_i$ values are non-negative for all $i \in V'$, by Theorem 1 we conclude 2^{nd} `best` is correct.

Since we have proved 2^{nd} `best` is correct when multiple nodes are simultaneously compromised and when nodes are compromised concurrent with any 2^{nd} `best` execution, we conclude that 2^{nd} `best` is correct when multiple nodes are compromised. ∎

Corollary 4. `purge` *is correct when a single node is compromised.*

Proof: Each $v \in adj(\overline{v})$ finds every destination, a, to which v's least cost path uses $\overline{v}$ as the first-hop node. v sets its least cost to each such a to ∞, thereby invalidating its path to a. v then initiates a diffusing computation. When an arbitrary node, i, receives a diffusing computation message from j, i iterates through each a specified in the message. If i routes via j to reach a, i sets its least cost to a to ∞, therefore invalidating any path to a with j and $\overline{v}$ an intermediate nodes.

By our assumptions, each node receives a diffusing computation message for each path using $\overline{v}$ as an intermediate node. Additionally, our assumptions imply that all diffusing computation terminate in finite time. Thus, we conclude that all paths using $\overline{v}$ as an intermediary node are invalidated in finite time.

Following the preprocessing, all $v \in adj(\overline{v})$ use distance vector to determine new least cost paths. Because all $dmatrix_i$ are non-negative for all $i \in V'$, by Theorem 1 we conclude that `purge` is correct. ∎

Corollary 5. `purge` *is correct when multiple nodes are compromised.*

Proof: The same proof used for Corollary 3 applies for `purge`. ∎

Corollary 6. `cpr` *is correct when a single node is compromised.*

Proof: `cpr` sets t' to the time $\overline{v}$ was compromised. Then, `cpr` rolls back using diffusing computations: each diffusing computation is initiated at each $v \in adj(\overline{v})$. Each node that receives a diffusing computation message, rolls back to a snapshot with timestep less than t'. By our assumptions, all nodes receive a message and the diffusing computation terminates in finite time. Thus, we conclude that each node $v \in V'$ rolls back to a snapshot with timestamp less than t' in finite time.

`cpr` then runs the preprocessing algorithm described in Section III-A, which removes each $\overline{v}$ as a destination in finite time (as shown in Corollary 2). Because each node rolls back to a snapshot in which all least costs are non-negative and `cpr` then uses distance vector to compute new least costs, by Theorem 1 we conclude that `cpr` is correct if no additional nodes are compromised during $[t', t^*]$. ∎

Corollary 7. `cpr` *is correct when multiple nodes are compromised.*

Proof: If multiple nodes, $\overline{V}$ are simultaneously compromised, `cpr` sets t' to the time the first $\overline{v} \in \overline{V}$ is compromised. Any nodes, $\overline{V}_2$, compromised concurrent with $\overline{V}$ (e.g., during $[t', t^*]$), trigger an additional `cpr` execution. The steps described in Section III-E ensure that all least cost computations (after rolling back) are to destination nodes $a \in V'$. By Theorem 1 we conclude `cpr` is correct because all $dmatrix_i$ are non-negative for all $i \in V'$. ∎

V. Analysis of Algorithms

In this section we first prove specific properties of our recovery algorithms (Section V-A) and then find communication complexity bounds for each recovery algorithm (Section V-B). All proofs assume a synchronous model in which nodes send and receive messages at fixed epochs. In each epoch, a node receives a message from all its neighbors and performs its local computation. In the next epoch, the node sends a message (if needed). Before we begin with the analysis, we introduce additional notation use in our proofs.

Notation. We use the definition of G and G' described in Section III. For convenience, $|V| = n$ and the diameter of G' is d. Let $\delta_t(i, j)$ be the least cost between nodes i and j – used by node i – at time t (we refer to this cost as $\delta(i,j)$). $p_t(i,j)$ refers to i's actual least cost path to j at time t. $p_s(i,j)$ is the least cost path from node i to j used by i at the start of recovery and $\delta_s(i,j)$ is the cost of this path; $p_w(i,j)$ is i's least cost path to j at time $t \in [t_b, t^*]$ and $\delta_w(i,j)$ the cost of this path [7]; and $p_f(i,j)$ is i's final least cost path to j (least cost at t^*) and has cost $\delta_f(i,j)$. $\ell(i,j)$ is the minimum number of links between nodes i and j in G'. Let $\max\limits_{i \in V}(|adj(i)|) = m$.

For each algorithm, let $\hat{t}$ mark the time all diffusing computations complete. Recall with `purge`, $\overline{v}$ is removed as a destination and $\overrightarrow{bad}$ state is invalidated in the *same* diffusing computations. Likewise, each `cpr` diffusing computation performs two actions: the diffusing computations remove $\overline{v}$ as a destination *and* implement the rollback. For this reason, $\hat{t}$ marks the same time across all three recovery algorithms. Let $C(i,j) = \delta_f(i,j) - \delta_{\hat{t}}(i,j)$. That is, $C(i,j)$ refers to the magnitude of change in $\delta(i,j)$ after the diffusing computations for each algorithm complete.

A. Properties of Recovery Algorithms

In this section we formally characterize how $\overrightarrow{min}$ values change during recovery. The properties established in this section will aid in understanding the simulation results presented in Section VI. Our proofs assume that link costs remain fixed during recovery (i.e., during $[t', t_b]$). We prove properties about $\overrightarrow{min}$ in order provide a precise characterization of recovery trends. In particular, our proofs establish that:

- The least cost between two nodes at the start of recovery is less than or equal to the least cost when recovery has completed. (Theorem 8)
- Before recovery begins, if the least cost between two nodes is less than its cost when recovery is complete, the path must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively. (Corollary 9)
- During 2nd `best` and `cpr` recovery, if the least cost between two nodes is less than its distance when recovery is complete, the path must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively. (Corollary 10)

The first two statements apply to any recovery algorithm because they make no claims about $\overrightarrow{min}$ values during recovery.

Theorem 8. $\forall i,j \in V'$, $\delta_s(i,j) \leq \delta_f(i,j)$

Proof: Assume $\exists i,j \in V'$ such that $\delta_s(i,j) > \delta_f(i,j)$. The paths available at the start of recovery are a superset of those available when recovery is complete. This means $p_f(i,j)$ is available before recovery begins. Distance vector would use this path rather than $p_s(i,j)$, implying that $\delta_s(i,j) = \delta_f(i,j)$, a contradiction. ■

Corollary 9. *$\forall i,j \in V'$, if $\delta_s(i,j) < \delta_f(i,j)$, then $p_s(i,j)$ is using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively.*

Proof: $\exists i,j \in V$ such that a path $p_s(i,j)$ with cost $\delta_s(i,j)$ is used before recovery begins where $\delta_s(i,j) < \delta_f(i,j)$ and $p_s(i,j)$ does not use $\overrightarrow{bad}$ or $\overrightarrow{old}$. The only paths available before recovery begins, which do not exist when recovery completes, are ones using $\overrightarrow{bad}$ or $\overrightarrow{old}$. Therefore, $p_s(i,j)$ must be available after recovery completes since we have assumed that $p_s(i,j)$ does not use $\overrightarrow{bad}$ or $\overrightarrow{old}$. Distance vector would use $p_s(i,j)$ instead of $p_f(i,j)$ because $\delta_s(i,j) < \delta_f(i,j)$. However this would imply that $\delta_s(i,j) = \delta_f(i,j)$, a contradiction. ■

[7]$p_w(i,j)$ and $\delta_w(i,j)$ can change during $[t_b, t^*]$.

Corollary 10. *For 2nd `best` and `cpr`. $\forall i,j \in V'$, if $\delta_w(i,j) < \delta_f(i,j)$ then $p_w(i,j)$ must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively.* [8]

Proof: We can use the same proof for Corollary 9 if we substitute $\delta_w(i,j)$ for $\delta_s(i,j)$ and $p_w(i,j)$ for $p_s(i,j)$. ■

Corollary 10 implies that 2nd `best` and `cpr` (after rolling back), count up from their initial costs – using $\overrightarrow{bad}$ or $\overrightarrow{old}$ state – until reaching the final correct least cost.

B. Communication Complexity

Next, we derive communication complexity bounds for each recovery algorithm. First, we consider graphs where link costs remain fixed (Section V-B1 - V-B4). Then, we derive bounds where link costs can change (Section V-B5).

We make the following assumptions in our complexity analysis:

- There is only a single compromised node, $\overline{v}$.
- We assume all nodes have unit link cost of 1 and that $\overline{v}$ falsely claims a cost of 1 to each $j \in V'$ (e.g., $\forall j \in V', \delta_s(v,j) = 1$).
- Since we assume unit link costs of 1, a link cost increase correspond to the removal of a link and a link cost decrease corresponds to the addition of a link.

1) Diffusing Computation Analysis: We begin our complexity analysis with a study of the diffusing computations common to all three of our recovery algorithms: 2nd `best`, `cpr`, and `purge`. In our analysis, we refer to a as our generic destination node.

Lemma 11. *Each diffusing computation has $O(E)$ message complexity.*

Proof: Each node in a diffusing computation sends a query to all downstream nodes and a reply to its parent node. Thus, no more than 2 messages are sent across a single edge, yielding $O(E)$ message complexity. ■

Theorem 12. *The diffusing computations for 2nd `best`, `cpr`, and `purge` have $O(mE)$ communication complexity.*

Proof: For each algorithm, diffusing computations are initiated at each $i \in adj(\overline{v})$, so there can be at most m diffusing computations. From Lemma 11, each diffusing computation has $O(E)$ communication complexity, yielding $O(mE)$ communication complexity. ■

[8]Corollary 10 does not apply to `purge` recovery because the $\delta_w(i,j) < \delta_w(i,j)$ condition is not always satisfied.

2) 2^{nd} `best` Analysis: Johnson [16] studies DV over topologies with bidirectional links and unit link costs of 1. Specifically, Johnson analyzes DV update activity after the failure of a single network resource, in which a resource is either a node or a link. She assumes that nodes adjacent to a failed resource detect the failure and then react according to DV: in the case of a failed node, each node sets its distance to the failed node to n and no link connected to the failed node is used in the final correct shortest paths. [9] From this point, DV behaves exactly like 2^{nd} `best`. [10] Therefore, by mapping our false path problem to Johnson's failed resource problem, we can use Johnson's analysis of DV to find bounds (and exact message counts) for 2^{nd} `best`. To do so, we modify the graph, G, that Johnson considers by adding false paths between $\overline{v}$ and all other nodes.

In Corollary 10, we proved that with 2^{nd} `best` nodes using $\overline{v}$ as an intermediate node count up from an initial incorrect least costs to their final correct value. Johnson proves the same for DV. Using this pattern, Theorem 13 derives upper and lower bounds for 2^{nd} `best`. Intuitively, the lower bound occurs when nodes count up by 2 (to their final correct value) and the upper bound results when nodes count up by 1.

Theorem 13. *After $\hat{t}$, 2^{nd} `best` message complexity is bounded below by*

$$\sum_{i \in V'} \left\lceil \frac{\max_{j \in V', i \neq j} (C(i,j))}{2} \right\rceil adj(i) \tag{1}$$

and above by

$$\sum_{i \in V'} \max_{j \in V', i \neq j} (C(i,j))\, adj(i) \tag{2}$$

Proof: Theorem 2 from [16] gives a lower bound of $\sum_{i,j \in V', i \neq j} \left\lceil \frac{1}{2} C(i,j) \right\rceil adj(i)$. However, this lower bound applies to a version of DV in which each message contains update costs for only a single destination; in a single epoch, if a node finds new least costs to multiple destinations, a separate message is sent for each destination with a new least cost (and is sent to each of the node's neighbors). In contrast, 2^{nd} `best` handles updates to multiple destinations concurrently: in each epoch, a single message sent by node i contains new distance values for all destinations in which i has a new least cost. For this reason, the maximum $C(i,j)$ value determines the number of times a node sends a message to each neighbor node.

The upper bound (Equation 2) is also derived from Theorem 2 in [16]. Theorem 2 gives us a upper bound of $\sum_{i,j \in V', i \neq j} C(i,j) \cdot adj(i)$. For the same reason described for the lower bound, the maximum $C(i,j)$ value determines the number of times a node sends a message to each neighbor node. ■

[9]The maximum distance to any node under Johnson's model is n, where n is the number of nodes in the graph. This is equivalent to ∞ in our case.

[10]Note that in contrast to Johnson, we assume an outside algorithm identifies the compromised node.

Corollary 14. *2^{nd} `best` has $O(mnd)$ communication complexity.*

Proof: From Lemma 12, 2^{nd} `best`'s diffusing computations have $O(mE)$ communication complexity. Next, 2^{nd} `best` runs DV. It must be the case that $C(i,j) \leq d$ and each node can at most have m neighbors. Since $|V'| = n - 1$, DV and therefore 2^{nd} `best` has $O(mnd)$ communication complexity. ■

Next, we restate Theorem 1 from Johnson [16] using our notation. Theorem 15 introduces the term *allowable path*. An allowable path from node i to $\overline{v}$ is a path in the original network (G) from node i to $\overline{v}$ which does not use $\overline{v}$ as an intermediate node.

Theorem 15. *Each incorrect route table entry assumes all possible lengths of paths of the form $|P| + \delta_s(\overline{v}, a)$ where P is an allowable path from node i to $\overline{v}$ and $\delta_s(\overline{v}, a)$ is the length of the false path claimed by $\overline{v}$.*

Theorem 15 translates the problem of finding the number of update messages after false node detection into the problem of finding all possible allowable paths between each node i and $\overline{v}$. By doing so, we can find the exact number of messages required for 2^{nd} `best` recovery.

The next two theorems, Theorem 16 and 17, follow from Theorem 5 in [16] and Theorem 15.

Theorem 16. *If G contains no odd cycles, the number of update messages after $\hat{t}$ is described exactly by Equation 1.*

Define $S(p)$ to be the set of nodes such that if $i \in S(p)$ there exists an allowable path of length p and $p+1$ from i to $\overline{v}$. Let $q(\overline{v}, i)$ be the smallest positive integer p such that $i \in S(p)$ and $q(\overline{v}, i) = c$.

Theorem 17. *If G contains an odd cycle and $c + \delta_s(\overline{v}, a) < \delta_f(i, a)$, then allowable paths to $\overline{v}$ increase in length by increments of 2 until reaching the value c and then increments by 1 thereafter. Thus, the number of changes in $\delta(i, a)$, after $\hat{t}$, is:*

$$C(i,a) - \frac{1}{2}\left(c - \delta_s(i,a)\right) \tag{3}$$

If $c + \delta_s(\overline{v}, a) \geq \delta_f(i, a)$, then update activity ceases before node i's least cost entries begin to increase by 1. Thus, in this case the number of update messages, after $\hat{t}$, is described exactly by Equation 1.

Theorem 15 tells us that before converging on the correct distance to a destination, a, 2^{nd} `best` exhaustively searches all paths from i to $\overline{v}$ and then uses $\overline{v}$'s false path to a. If G contains no odd cycle, then i counts up by 2 until reaching the final correct cost to a. Node i does so by hopping back and forth between an adjacent node j (where $j \neq \overline{v}$) k times (for some integer $k \geq 0$), then uses an allowable path from i to $\overline{v}$, and finally uses $\overline{v}$'s false path to a.

However, if G contains an odd cycle then the update behavior is slightly more complicated. Node i counts up by 2 until $\delta(i, a)$ reaches a specific value, c^*, at which point, i counts up

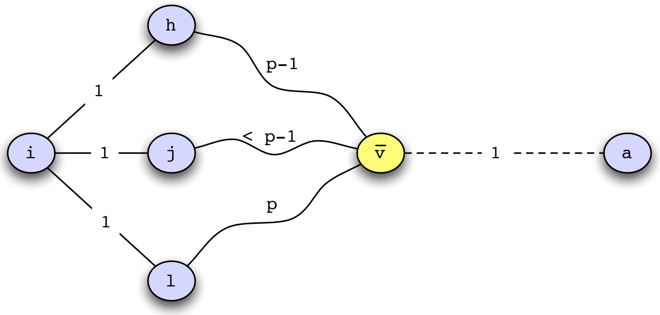


Fig. 3. The yellow node ($\overline{v}$) is the compromised node. The dotted line from $\overline{v}$ to a represents the false path.

by 1 until i converges on the final correct distance to a. In Figure 3, $c^* = \delta(i,h)+\delta(h,\overline{v})+\delta_s(\overline{v},a) = 1+(p-1)+1 = p+1$. In the epoch after $\delta(i,a)$ is set to c^*, node i uses its path via h of length p to $\overline{v}$ (and then $\overline{v}$'s false path to a). In the following epoch, i uses its path via l of length $p+1$ to $\overline{v}$. From this point, i counts up by 1 by using allowable paths of lengths $p+2k$, for integer $k \geq 1$, (by hopping back and forth between h) to $\overline{v}$ and allowable paths of length $(p+1)+2k$ (by ping-ponging with l) to $\overline{v}$, until $\delta(i,a)$ counts up to $\delta_f(i,j)$.

3) `cpr` *Analysis:* The analysis for 2nd `best` applies to `cpr` because after rolling back `cpr`, executes the steps of 2nd `best`. In fact, because `cpr` performs the rollback using the same diffusing computations analyzed for 2nd `best` (e.g., the diffusing computations that remove $\overline{v}$ as a destination), the results for 2nd `best` apply to `cpr` with no changes.

Although Theorem 13, Theorem 16, and Theorem 17 apply directly to `cpr`, the bounds and exact message count can defer between 2nd `best` and `cpr`. In most cases, $\delta_{\hat{t}}(i,j)$ for 2nd `best` is smaller than $\delta_{\hat{t}}(i,j)$ for `cpr` because `cpr` rolls back to a checkpoint taken before $\overline{v}$ is compromised. [11] Thus, `cpr`'s $C(i,j)$ values are typically smaller than those for 2nd `best`, resulting in lower message complexity for `cpr`.

4) `purge` *Analysis:* Our `purge` analysis establishes that after the diffusing computations complete, all nodes using false routing state to reach a destination have a least cost of ∞ to this destination. From this point, these least costs remain ∞ until updates from nodes with a non-infinite cost to the destination spread through the network. Upon receiving a non-infinite least cost to the destination, nodes switch from an infinite least cost to a finite one (Lemma 18). We establish that the first finite cost to the destination is in fact the node's final correct least cost to the destination (Theorem 20). In this way, least costs change from ∞ to their final correct value.

In the presence of a tie, we assume a node uses the least cost path that avoids $\overline{v}$. Note that if ties are broken by using the path with $\overline{v}$ as intermediate node, our proofs still apply, although with a few minor changes. Now we are ready to define two sets that are key structures in our `purge` proofs.

[11]At worst, $\delta_{\hat{t}}(i,j)$ is equivalent across 2nd `best` and `cpr`. This occurs when the false least vector claimed by $\overline{v}$ matches the least cost vector used by $\overline{v}$ before being compromised (e.g., $\overrightarrow{bad}$=$\overrightarrow{old}$).

Definition 2. *Let $B(a,t)$ be the set of nodes that have least cost ∞ to destination node a at time t.*

Definition 3. *$F(a,t)$ is the set of nodes such that if $b \in F(a,t)$ then the following must be true:*

1) $b \notin B(a,t)$.
2) $\exists b' : b' \in adj(b) \wedge b' \notin B(a,t)$.
3) $\exists b'' : b'' \in adj(b) \wedge b'' \in B(a,t)$.

Next, in Lemma 18 we prove that the size of $B(a,t)$ shrinks by at least one for each timestep beginning with t'' – where t'' refers to the time that the first $i \in V'$ with $\delta(i,a) = \infty$ changes $\delta(i,a)$ to a finite value – until $B(a,t)$ is empty.

Lemma 18. *For each $t \geq t''$, $|B(a,t)| \geq |B(a,t+1)|+1$, until $B(a,t) = \emptyset$.*

Proof: Once `purge` diffusing computations complete at $\hat{t}$, a DV computation is triggered at each $v \in adj(\overline{v})$. At this point, all least costs corresponding to paths using $\overline{v}$ as an intermediate node are set to ∞ (this is proved in Corollary 4). As such, each $i \in B(a,\hat{t})$ sends a DV message with a least of ∞ to each neighbor, [12] unless i has a neighbor node in $F(a,\hat{t})$ (note that we denote this time as t''). In this case, i selects a finite least to a (which implies $i \notin B(a,t'')$), triggering the propagation of finite least costs to a. Specifically, in each subsequent timestep t (until $B(a,t) = \emptyset$) at least one node, j, changes $\delta_t(j,a)$ from ∞ to a finite value. This is the case because unless $B(a,t) = \emptyset$, a node i that has changed $\delta_t(i,a)$ from ∞ to a finite value, has $j \in adj(i)$ with $\delta_t(j,a) = \infty$ and thus $\delta_{t+1}(j,a)$ will be finite. A finite $\delta_{t+1}(j,a)$ value implies $j \notin B(a,t+1)$. Since $B(a,t)$ is monotonic, eventually $B(a,t) = \emptyset$. ■

Our next Lemma (19) lists all possible values for the number of links between any $b \in F(a,\hat{t})$ and $\overline{v}$. We later use this Lemma in Theorem 20.

Lemma 19. *For all $b \in F(a,\hat{t})$, $\ell(b,\overline{v}) = \{\ell(b,a), \ell(b,a)-1\}$.*

Proof: Let b be an arbitrary node in $F(a,\hat{t})$. If $\ell(b,\overline{v}) < \ell(b,a)-1$, this would imply $b \in B(a,\hat{t})$, a contradiction (a violation of condition 1 of the $F(a,\hat{t})$ definition). On the other hand, consider the case where $\ell(b,\overline{v}) > \ell(b,a)$ and where $b' \in adj(b)$ and $b' \in B(a,\hat{t})$. Any path b' uses with $\overline{v}$ as an intermediate node has cost $\ell(b,\overline{v}) - 1 + \delta_s(\overline{v},a) = \ell(b,\overline{v}) - 1 + 1 = \ell(b,\overline{v})$. Since we have assumed $\ell(b,\overline{v}) > \ell(b,a)$, b' would use b as a next-hop router along $p_{\hat{t}}(b',a)$. This implies $b' \notin B(a,\hat{t})$, a contradiction. ■

The following theorem is the key argument in establishing `purge`'s communication complexity. Theorem 20 proves that once any $i \in V'$ changes its least cost from ∞, i changes its least cost to the final correct value.

Theorem 20. *For $t > \hat{t}$ and an arbitrary destination $a \in V'$, each $i \in B(a,\hat{t})$ with $\delta_{\hat{t}}(i,a) = \infty$ only modifies $\delta(i,a)$ once, such that $\delta(i,a)$ changes from ∞ to $\delta_f(i,a)$.*

[12]Recall that after $\hat{t}$, `purge` forces each node to send a least cost message to each neighbor (even if the node's least cost has not changed since $\hat{t}$).

Proof: Consider an arbitrary $i \in V'$ such that $i \in B(a,\hat{t})$. i must use some $b \in F(a,\hat{t})$ as an intermediate node along $p_f(i,a)$. Let b^* be this node. If we show that $\delta_f(b^*,a)$ is the first least cost among all $b \in F(a,\hat{t})$ to reach i, then we have proved our claim because in Lemma 18 we proved that i does not update its least cost to a finite value until it receives a least cost from a $b \in F(a,\hat{t})$. [13] For the sake of contradiction, assume that for some $b' \in F(a,\hat{t})$, where $b' \neq b^*$, that $\delta_f(b',a)$ reaches i before $\delta_f(b^*,a)$. [14] This implies that:

$$\ell(b',\overline{v}) + \ell(i,b') \quad < \quad \ell(b^*,\overline{v}) + \ell(i,b^*) \qquad (4)$$

From Lemma 19, we know that $\ell(b',\overline{v}) = \{\ell(b',a), \ell(b',a)-1\}$ and $\ell(b^*,\overline{v}) = \{\ell(b^*,a), \ell(b^*,a)-1\}$ If we substitute $\ell(b',\overline{v}) = \ell(b',a)$ and $\ell(b^*,\overline{v}) = \ell(b^*,a)$ into Equation 4, it yields:

$$\ell(b',a) + \ell(i,b') \quad < \quad \ell(b^*,a) + \ell(i,b^*) \qquad (5)$$

However, since we have assumed that i routes via b^*, we know that:

$$\ell(b',a) + \ell(i,b') \quad > \quad \ell(b^*,a) + \ell(i,b^*) \qquad (6)$$

Thus, between Equation 5 and Equation 6 we have a contradiction. Similar contradictions can be derived by substituting all other permutations of the $\ell(b',\overline{v})$ and $\ell(b^*,\overline{v})$ equalities, derived from Lemma 19. In conclusion, we have shown by contradiction that $\delta(i,a)$ only changes a single time: $\delta(i,a)$ changes from ∞ to $\delta_f(i,a)$. ■

Corollary 21. `purge` *is loop-free at every instant of time.*

Proof: Before $\hat{t}$, only diffusing computation run. Diffusing computations are loop-free because computation proceeds along spanning trees, which are by definition acyclic. After $\hat{t}$, only DV computations run. From Theorem 20 we know that each node with least cost ∞ to an arbitrary destination, changes its least cost once: from ∞ to the correct final least cost. We conclude that `purge` is loop free. ■

Theorem 22. `purge` *message complexity is* $O(mnd)$.

Proof: `purge` consists of two steps: the diffusing computations to invalidate false state and DV to compute new least cost paths invalidated by the diffusing computations. From Lemma 12, `purge`'s diffusing computations have $O(mE)$ communication complexity. The DV message complexity can be understood as follows. To start the computation, `purge` enforces that each node sends DV message (to each neighbor), even if no least costs are found. From Theorem 20 and Lemma 18, all $i \in B(a,\hat{t})$ only change $\delta(i,a)$ once: $\delta(i,a)$ changes from ∞ to $\delta_f(i,a)$. `purge` computations to all destinations run in parallel, meaning that all least cost updates to nodes h away are handled in the same round of update messages. For this reason, `purge` only sends messages $d+1$ times after $\hat{t}$. Finally, since there are $n-1$ nodes, each with a maximum of m neighbors, and each node sends messages $d+1$ times, `purge` communication complexity if $O(mnd)$. ■

[13]Note that any node i with $\delta(i,a) = \infty$ only changes $\delta(i,a)$ to a finite value. Thus, when `purge` forces nodes to send a message after $\hat{t}$ to initiate the DV computation, no $i \in B(a,\hat{t})$ receiving a least cost of ∞ updates its least cost.

[14]From Lemma 18 we know that a finite least cost to a reaches every node in $B(a,\hat{t})$.

5) Analysis with Link Cost Changes: In this section, we analyze each of our algorithms in the case where w link cost changes occur. Because we assume unit link costs of 1, a link cost decrease corresponds to the addition of a new link and a link cost increase corresponds to the removal of a link. In our analysis, we assume that all w link cost changes finish propagating before $\overline{v}$ is detected (e.g., before t_b).

The analysis for 2nd `best` and `purge` from Section V-B2 and Section V-B4, respectively, does not change. This is the case because 2nd `best` and `purge` do not roll back in time, and thus all w link cost changes are accounted for when recovery begins at t_b. The `cpr` analysis from Section V-B3 changes because after rolling back, all w link cost changes need to be replayed.

Let $\delta'_f(i,a)$ be node i's final least cost to a if no link cost changes occur during $[t', t_b]$. Define $C'(i,j) = \delta'_f(i,a) - \delta_{\hat{t}}(i,j)$.

The communication complexity for a link cost increase is $O(n^2)$ [16] and $O(E)$ for a link cost decrease [15]. Let there be u link cost increases (e.g., u links are removed from G) and $w-u$ link cost decreases (e.g., $w-u$ links are added to G). At worst, the link cost changes are processed after $\overline{v}$ recovery completes. As a result, `cpr` communication complexity with link cost changes is bounded above by:

$$\sum_{i \in V'} \max_{j \in V', i \neq j} \left(C'(i,j)\right) adj(i) + O(un^2) + O\left((w-u)E\right) \quad (7)$$

6) Discussion: The communication complexity for 2nd `best`, `cpr`, and `purge` are all $O(mnd)$ over graphs with fixed unit link costs. It is not surprising that the communication complexity is the same because all three algorithms use DV as their final step and DV asymptotically dominates the communication complexity of each recovery algorithm. In this context, the differing performance of our three algorithms that we found in our simulations is determined by the hidden constants.

We also bounded the communication overhead incurred by `cpr` under conditions of link cost changes. This overhead is not incurred by 2nd `best` and `purge` because do not roll back in time, and thus all link cost changes are accounted for when recovery begins.

VI. Evaluation

In this section, we use simulations to characterize the performance of each of our three recovery algorithms in terms of message and time overhead. Our goal is to illustrate the relative performance of our recovery algorithms over different topology types (e.g., Erdös-Rényi graphs, Internet-like graphs) and different network conditions (e.g., fixed link costs, changing link costs).

We build a custom simulator with a synchronous communication model as described in Section V. All algorithms are deterministic under this communication model. The synchronous communication model, although simple, yields interesting insights into the performance of each of the recovery algorithms. Evaluation of our algorithms using a more general asynchronous communication model is currently under investigation. However, we believe an asynchronous implementation will demonstrate similar trends.

We simulate the following scenario: [15]

1) Before t', $\forall v \in V$ $\overrightarrow{min_v}$ and $dmatrix_v$ are correctly computed.
2) At time t', $\overline{v}$ is compromised and advertises a $\overrightarrow{bad}$ (a vector with a cost of 1 to *every* node in the network) to its neighboring nodes.
3) $\overrightarrow{bad}$ spreads for a specified number of hops (this varies by experiment). Variable k refers to the number of hops that $\overrightarrow{bad}$ has spread.
4) At time t, some node $v \in V$ notifies all $v \in adj(\overline{v})$ that $\overline{v}$ was compromised. [16]

The message and time overhead are measured in step (4) above. The pre-computation common to all three recovery algorithms, described in Section III-A, is not counted towards message and time overhead. We describe our simulation scenario for multiple compromised nodes in Section VI-A4.

A. Fixed Link Weight Experiments

In the next five experiments, we evaluate our recovery algorithms over different topology types in the case of fixed link costs.

1) Experiment 1 - Erdös-Rényi Graphs with Fixed Unit Link Weights: We start with a simplified setting and consider Erdös-Rényi graphs with parameters n and p. n is the number of graph nodes and p is the probability that link (i, j) exists where $i, j \in V$. The link weight of each edge in the graph is set to 50. We iterate over different values of k. For each k, we generate an Erdös-Rényi graph, $G = (V, E)$, with parameters n and p. Then we select a $\overline{v} \in V$ uniformly at random and simulate the scenario described above, using $\overline{v}$ as the compromised node. In total we sample 20 unique nodes for each G. We set $n = 100$, $p = \{0.05, 0.15, 0.25, 0.50\}$, and let $k = \{1, 2, ...10\}$. Each data point is an average over 600 runs (20 runs over 30 topologies). We then plot the 90% confidence interval.

For each of our recovery algorithms, Figure 4 shows the message overhead for different values of k. We conclude that `cpr` outperforms `purge` and 2nd `best` across all topologies. `cpr` performs well because $\overrightarrow{bad}$ is removed using a single diffusing computation, while the other algorithms remove $\overrightarrow{bad}$ state through distance vector's iterative process. `cpr`'s global state after rolling back is almost the same as the final recovered state.

2nd `best` recovery can be understood as follows. By Corollary 9 and 10 in Section V-A, distance values increase

	$k = 1$	$k = 2$	$k = 3$	$k = 4 - 10$
$p = 0.05$	0	14	87	92
$p = 0.15$	0	7	8	9
$p = 0.25$	0	0	0	0
$p = 0.50$	0	0	0	0

TABLE II
AVERAGE NUMBER PAIRWISE ROUTING LOOPS FOR 2nd `best` IN EXPERIMENT 1.

	$k = 1$	$k = 2$	$k = 3$	$k = 4 - 10$
$p = 0.05$	554	1303	9239	12641
$p = 0.15$	319	698	5514	7935
$p = 0.25$	280	446	3510	5440
$p = 0.50$	114	234	2063	2892

TABLE III
AVERAGE NUMBER PAIRWISE ROUTING LOOPS FOR 2nd `best` IN EXPERIMENT 2.

from their initial value until they reach their final (correct) value. Any intermediate, non-final, distance value uses $\overrightarrow{bad}$ or $\overrightarrow{old}$. Because $\overrightarrow{bad}$ and $\overrightarrow{old}$ no longer exist during recovery, these intermediate values must correspond to routing loops. Table II shows that there are few pairwise routing loops during 2nd `best` recovery in the network scenarios generated in Experiment 1, indicating that 2nd `best` distance values quickly count up to their final value. [17] Although no pairwise routing loops exist during `purge` recovery, `purge` incurs overhead in performing network-wide state invalidation. Roughly, 50% of `purge`'s messages come from these diffusing computations. For these reasons, `purge` has higher message overhead than 2nd `best`.

Figure 5 shows the time overhead for the same p values. The trends for time overhead match the trends we observe for message overhead. [18]

`purge` and 2nd `best` message overhead increases with larger k. Larger k imply that false state has propagated further in the network, implying more paths to repair, and therefore increased messaging. For values of k greater than a graph's diameter, the message overhead remains constant, as expected.

2) Experiment 2 - Erdös-Rényi Graphs with Fixed but Randomly Chosen Link Weights: The experimental setup is identical to Experiment 1 with one exception: link weights are selected uniformly at random between $[1, n]$ (rather than using fixed link weight of 50).

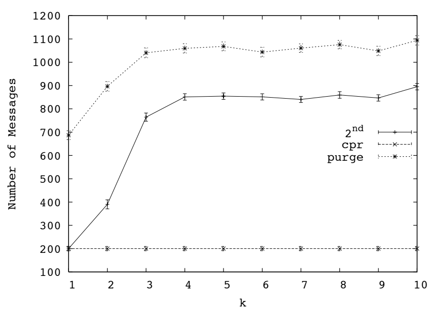
Figure 6 show the message overhead for different k where $p = \{0.05, 0.15, 0.25, 0.50\}$. In striking contrast to Experiment 1, `purge` outperforms 2nd `best` for most values of k. 2nd `best` performs poorly because the count-to-∞ problem: Table III shows the large average number of pairwise routing loops in this experiment, an indicator of the occurrence of count-

[15]In Section VI-A4 we consider the case of multiple compromised nodes. In that experiment we modify our simulation scenario to consider a set of compromised nodes, $\overline{V}$, instead of $\overline{v}$.

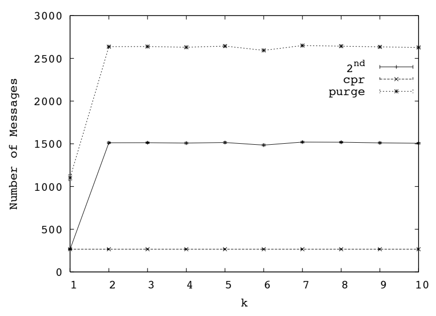
[16]For `cpr` this node also indicates the time, t', $\overline{v}$ was compromised.

[17]We compute this metric as follows. After each simulation timestep, we count all pairwise routing loops over all source-destination pairs and then sum all of these values.

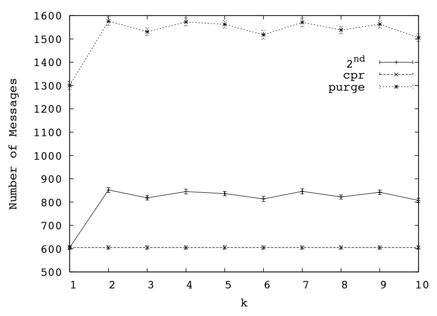
[18]For the remaining experiments, we omit time overhead plots because time overhead follows the same trends as message overhead.



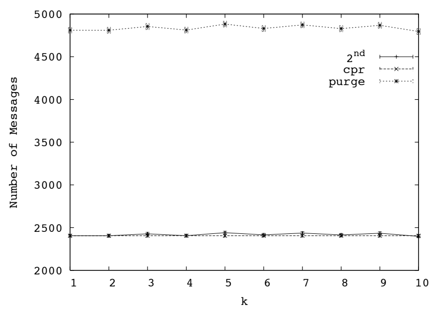
(a) $p = 0.05$, diameter=6.14



(b) $p = 0.15$, diameter=3.01



(c) $p = 0.25$, diameter=2.99



(d) $p = 0.50$, diameter=2

Fig. 4. Experiment 1: message overhead for Erdös-Rényi Graphs with Fixed Unit Link Weights generated over different p values. Note the y-axis have different scales.

to-∞ problem. In the few cases (e.g., $k = 1$ for $p = 0.15$, $p = 0.25$ and $p = 0.50$) that 2nd `best` performs better than `purge`, 2nd `best` has few routing loops.

No routing loops are found with `purge`. `cpr` performs well for the same reasons described in Section VI-A1.

In addition, we counted the number of epochs in which at least one pairwise routing loop existed. For 2nd `best` (across all topologies), on average, all but the last three timesteps had at least one routing loop. This suggests that the count-to-∞ problem dominates the cost for 2nd `best`.

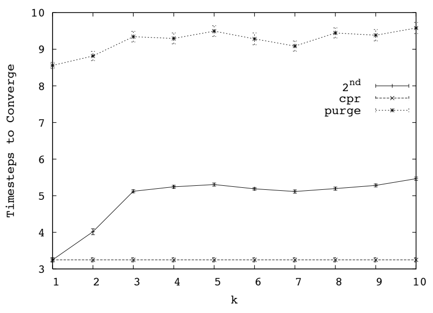
3) Experiment 3 - Internet-like Topologies: Thus far, we studied the performance of our recovery algorithms over Erdös-Rényi graphs, which have provided us with useful intuition about the performance of each algorithm. In this experiment, we simulate our algorithms over Internet-like topologies downloaded from the Rocketfuel website [3] and generated using GT-ITM [2]. The Rocketfuel topologies have inferred edge weights. For each Rocketfuel topology, we let each node be the compromised node and average over all of these cases for each value of k. For GT-ITM, we used the parameters specified in Heckmann et al [11] for the 154-node AT&T topology described in Section 4 of [11]. For the GT-ITM topologies, we use the same criteria specified in Experiment 1 to generate each data point.

The results, shown in Figure 7, follow the same pattern as in Experiment 2. In the cases where 2nd `best` performs poorly, the count-to-∞ problem dominates the cost, as evidenced by the number of pairwise routing loops. In the few cases that 2nd `best` performs better than `purge`, there are few pairwise routing loops.

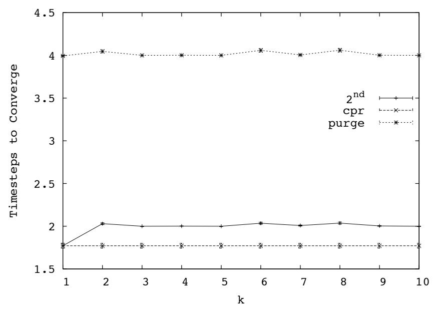
4) Experiment 4 - Multiple Compromised Nodes: In this experiment, we evaluate our recovery algorithms when multiple nodes are compromised. Our experimental setup is different from what we have used to this point: we fix $k = \infty$ and vary the number of compromised nodes. Specifically, for each topology we create $m = \{1, 2, ..., 15\}$ compromised nodes, each of which is selected uniformly at random (without replacement). We then simulate the scenario described at the start of Section VI with one modification: m nodes are compromised during $[t', t' + 10]$. The simulation is setup so that the outside algorithm identifies all m compromised node at time t. After running the simulation for all possible values for m, we generate a new topology and repeat the above procedure. We continue sampling topologies until the 90% confidence interval for message overhead falls within 10% of the mean message overhead.

First, we perform this experiment using Erdös-Rényi graphs with fixed link costs. The message overhead results are shown in Figure 8(a) for $p = 0.05$ and $n = 100$. [19] The relative performance of the three algorithms is consistent with the results from Experiment 1, in which we had a single compromised node. As in Experiment 1, 2nd `best` and `cpr` have few pairwise routing loops (Figure 8(b)). In fact, there is more than an order of magnitude fewer pairwise routing loops in this experiment when compared to the results for the same simulation scenario of m compromised nodes using Erdös-Rényi graphs with random link weights (Figure 9(b)). Few

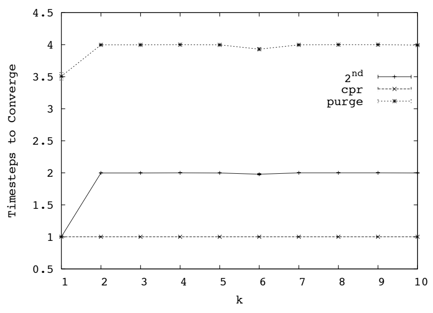
[19]We do not include the results for $p = \{0.15, 0.25, 0.50\}$ because they are consistent with the results for $p = 0.05$.



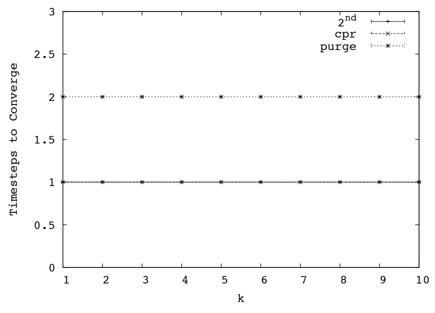
(a) $p = 0.05$, diameter=6.14



(b) $p = 0.15$, diameter=3.01



(c) $p = 0.25$, diameter=2.99



(d) $p = 0.50$, diameter=2

Fig. 5. Experiment 1: time overhead for Erdös-Rényi Graphs with Fixed Unit Link Weights generated over different p values.

routing loops imply that 2nd `best` and `cpr` (after rolling back) quickly count up to correct least costs. In contrast, `purge` has high message overhead because `purge` globally invalidates false state before computing new least cost paths, rather than directly using alternate paths that are immediately available when recovery begins at time t.

2nd `best` and `purge` message overhead are nearly constant for $m \geq 8$ because at that point $\overrightarrow{bad}$ state has saturated G. Figure 8 shows the number of least cost paths, per node, that use $\overrightarrow{bad}$ or $\overrightarrow{old}$ at time t (e.g., after $\overrightarrow{bad}$ state has propagated k hops from $\overline{v}$). The number of least cost paths that use $\overrightarrow{bad}$ is nearly constant for $m \geq 8$.

In contrast, `cpr` message overhead increases with the number of compromised nodes. After rolling back, `cpr` must remove all compromised nodes and all stale state (e.g., $\overrightarrow{old}$) associated with each $\overline{v}$. As seen in Figure 8(c), the amount of $\overrightarrow{old}$ state increases as the number of compromised nodes increase.

Next, we perform the same experiment using Erdös-Rényi graphs with with link weights selected uniformly at random from $[1, 100]$. We only show the results for $p = .05$ and $n = 100$ because the trends are consistent for other values of p. The message overhead results for this experiment are shown in Figure 9(a). `purge` performs best because, unlike 2nd `best` and `cpr`, `purge` does not suffer from the count-to-∞ problem. Below, we explain the performance of each algorithm in detail.

Consistent with Experiment 2 and 3, 2nd `best` performs poorly because of the count-to-∞ problem. Figure 9(b) shows that a significant number of pairwise routing loops occur during 2nd `best` recovery. 2nd `best` message overhead remains constant when $m \geq 6$ because at this point $\overrightarrow{bad}$ state has saturated the network. Figure 9(c) confirms this: the number of effected least cost paths remains constant (at 80) for all $m \geq 6$.

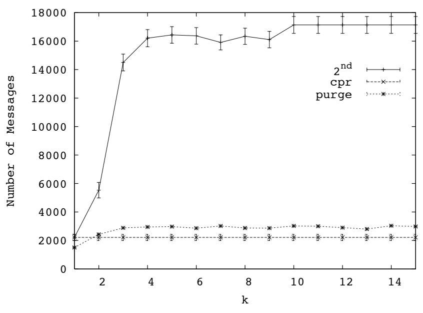
`cpr` message overhead increases with the number of compromised nodes because the amount of $\overrightarrow{old}$ state increases as the number of compromised nodes increase (Figure 9(c)). More $\overrightarrow{old}$ state results in more routing loops – as shown in Figure 9(b) – causing increased message overhead.

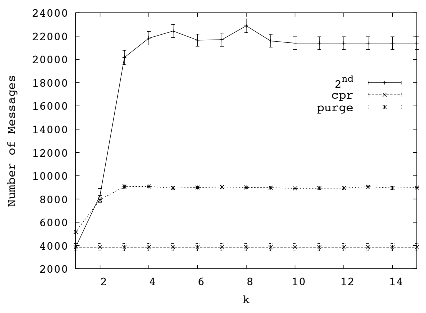
`purge` performs well because unlike `cpr` and 2nd `best`, no routing loops occur during recovery. Surprisingly, `purge`'s message overhead decreases when $m \geq 5$. Although more least cost paths need to be computed with larger m, the message overhead decreases because the residual graph, G', – resulting from the removal of all m compromised nodes – is smaller than G. As a result, there are m fewer destinations and m fewer nodes sending messages during the recovery process.

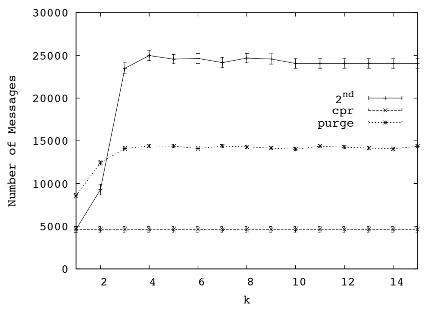
Finally, we simulated the same scenario of m compromised node using the Internet-like graphs from Experiment 3. The results were consistent with those for Erdös-Rényi graphs with random link weights.

5) Experiment 5 - Poison Reverse: We repeat Experiments 2, 3, and 4 using poison reverse for 2nd `best` and `cpr`. We do not apply poison reverse to `purge` because no routing loops (resulting from the removal of $\overline{v}$) exist during `purge`'s recovery. Additionally, we do not repeat Experiment 1 using poison reverse because we observed few routing loops in that experiment.

First, we repeat Experiment 2 using poison reverse. The


(a) $p = 0.05$, diameter=6.14


(b) $p = 0.15$, diameter=3.01


(c) $p = 0.25$, diameter=2.99

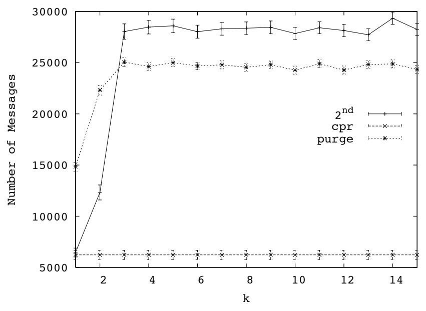
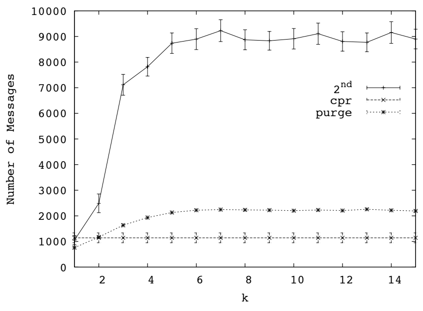
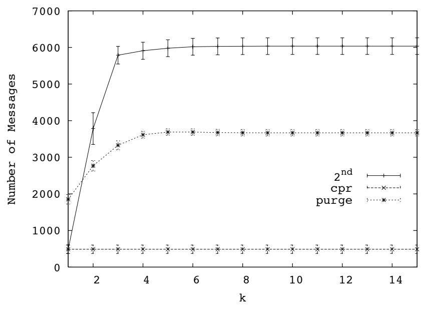

(d) $p = 0.50$, diameter=2

Fig. 6. Experiment 2: message overhead for Erdös-Rényi graph with link weights selected uniformly random from $[1, 100]$. Note the y-axis have different scales.


(a) GT-ITM, $n = 156$, diameter=14.133


(b) Rocketfuel 6461, $n = 141$, diameter=8

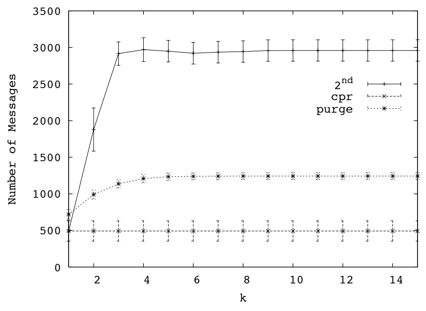

(c) Rocketfuel 3867, $n = 79$, diameter=10

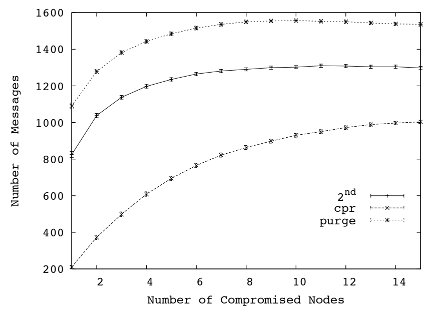
Fig. 7. Experiment 3: Internet-like graph message overhead

results are shown for one representative topology in Figure 10(a), where 2^{nd} best + pr and cpr + pr refer to each respective algorithm using poison reverse.

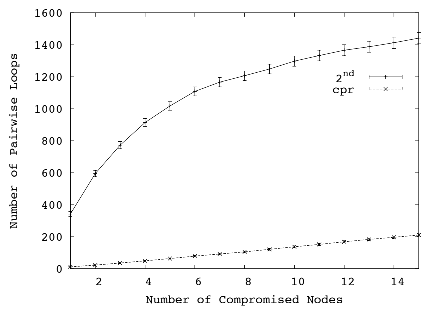
cpr + pr has modest gains over standard cpr because few routing loops occur with cpr. On other hand, 2^{nd} best + pr sees a significant decrease in message overhead when compared to the standard 2^{nd} best algorithm because poison reverse removes the many pairwise routing loops that occur during 2^{nd} best recovery. However, 2^{nd} best + pr still performs worse than cpr + pr and purge. When compared to cpr + pr, the same reasons described in Experiment 2 account for 2^{nd} best + pr's poor performance. Comparing purge and 2^{nd} best + pr yields interesting insights into the two different approaches for eliminating routing loops: purge prevents routing loops using diffusing computations and 2^{nd} best + pr uses poison reverse. Because purge has lower message complexity than 2^{nd} best + pr and poison reverse only eliminates pairwise routing loops, it suggests that purge removes routing loops larger than 2. We are currently investigating this claim.

Repeating Experiment 3 using poison reverse yields the same trends as repeating Experiment 2 with poison reverse. Finally, we consider poison reverse in the case of multiple compromised nodes (e.g., we repeat Experiment 4). 2^{nd} best + pr and cpr + pr over Erdös-Rényi graphs with unit link weights perform only slightly better than the basic version of each algorithm, respectively. This is expected because few pairwise routing loops occur in this scenario.

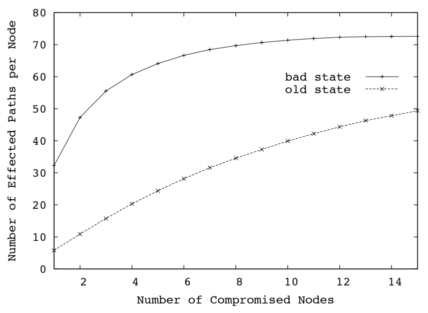
Like the single compromised node scenario, in the case of multiple compromised nodes, 2^{nd} best + pr and cpr + pr over Erdös-Rényi graphs with random link weights provide significant improvements over the basic version of



(a) Message Overhead

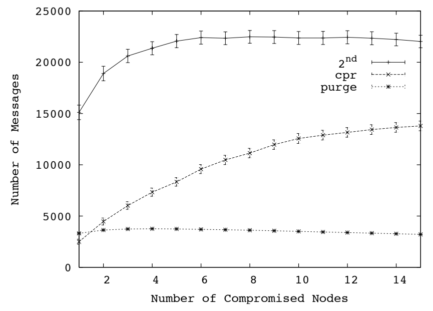


(b) Pairwise Routing Loops

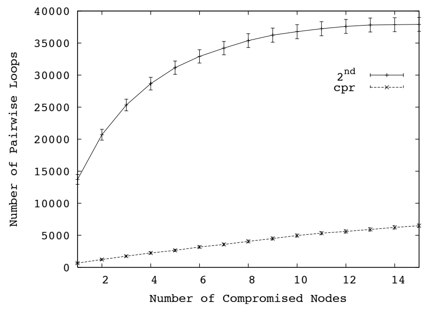


(c) Number of Effected Least Cost Paths

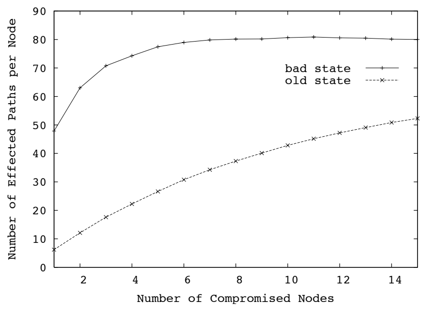
Fig. 8. Experiment 4 - multiple compromised nodes over Erdös-Rényi graphs with fixed link weights, $p = .05$, $n = 100$, and diameter=6.14.



(a) Message Overhead



(b) Pairwise Routing Loops



(c) Number of Effected Least Cost Paths

Fig. 9. Experiment 4 - multiple compromised nodes over Erdös-Rényi graphs with link weights selected uniformly at random from $[1, 100]$, $p = .05$, $n = 100$, and diameter=6.14.

each algorithm. Particularly for 2^{nd} `best`, we observed many pairwise loops in Experiment 4 (Figure 9(b)). This accounts for the effectiveness of poison reverse in this experiment. Despite the significant improvements, 2^{nd} `best` + `pr` still performs worse than `cpr` + `pr` and `purge`. `cpr` + `pr` performs best among all the recovery algorithms because, as we have discussed, rolling back to a network-wide checkpoint is more efficient than using distance vector's iterative procedure. Furthermore, poison reverse helps `cpr` + `pr` reduce the count-to-∞ problem, improving `cpr`'s effectiveness in the face of multiple compromised nodes.

B. Link Weight Change Experiments

So far, we have evaluated our algorithms over different topologies with fixed link costs in scenarios with single and multiple compromised nodes. We found that `cpr` using poison reverse outperforms the other algorithms because `cpr` removes false routing state with a single diffusing computation, rather than using an iterative distance vector process as in 2^{nd} `best` and `purge`, and poison reverse removes all pairwise routing loops that occur during `cpr` recovery.

In the next three experiments we evaluate our algorithms over graphs with changing link costs. We introduce link cost changes between the time $\overline{v}$ is compromised and when $\overline{v}$ is discovered (e.g., during $[t', t_b]$). In particular, let there be λ link cost changes per timestep, where λ is deterministic. To create a link cost change event, we choose a link (except for all $(v, \overline{v})$ links) whose link will change equiprobably among all links. The new link cost is selected uniformly at random from $[1, n]$.

1) Experiment 6 - Link Cost Changes: Except for λ, our experimental setup is identical to the one in Experiment 2. We let $\lambda = \{1, 4, 8\}$. In order to isolate the effects of link costs changes, we assume that `cpr` checkpoints at each timestep.

Figure 11 shows `purge` yields the lowest message overhead for $p = .05$, but only slightly lower than `cpr`. `cpr`'s message overhead increases with larger k because there are more link cost change events to process. After `cpr` rolls back, it must process all link cost changes that occurred in $[t', t_b]$. In contrast, 2^{nd} `best` and `purge` process some of the link cost change events during the interval $[t', t_b]$ as part of normal distance vector execution. In our experimental setup, these messages are not counted because they do not occur in Step 4 (i.e., as part of the recovery process) of our simulation scenario described in Section VI.

Our analysis further indicates that 2^{nd} `best` performance suffers because of the count-to-∞ problem. The gap between 2^{nd} `best` and the other algorithms shrinks as λ increases because as λ increases, link cost changes have a larger effect on message overhead.

With larger p values, λ has a smaller effect on message complexity because more alternate paths are available. Thus when $p = 0.15$ and $\lambda = 1$, most of `purge`'s recovery effort is towards removing $\overrightarrow{bad}$ state, rather than processing link cost changes. Because `cpr` removes $\overrightarrow{bad}$ using a single diffusing computation and there are few link cost changes,

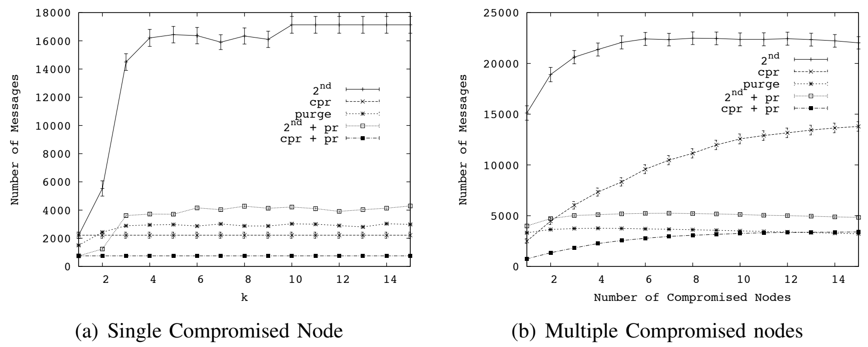


Fig. 10. Experiment 5 plots. Algorithms run over Erdös-Rényi graphs with random link weights, $n = 100$, $p = .05$, and average diameter=6.14. 2^{nd} best + pr refers to 2^{nd} best using poison reverse. Likewise, cpr + pr is cpr using poison reverse.

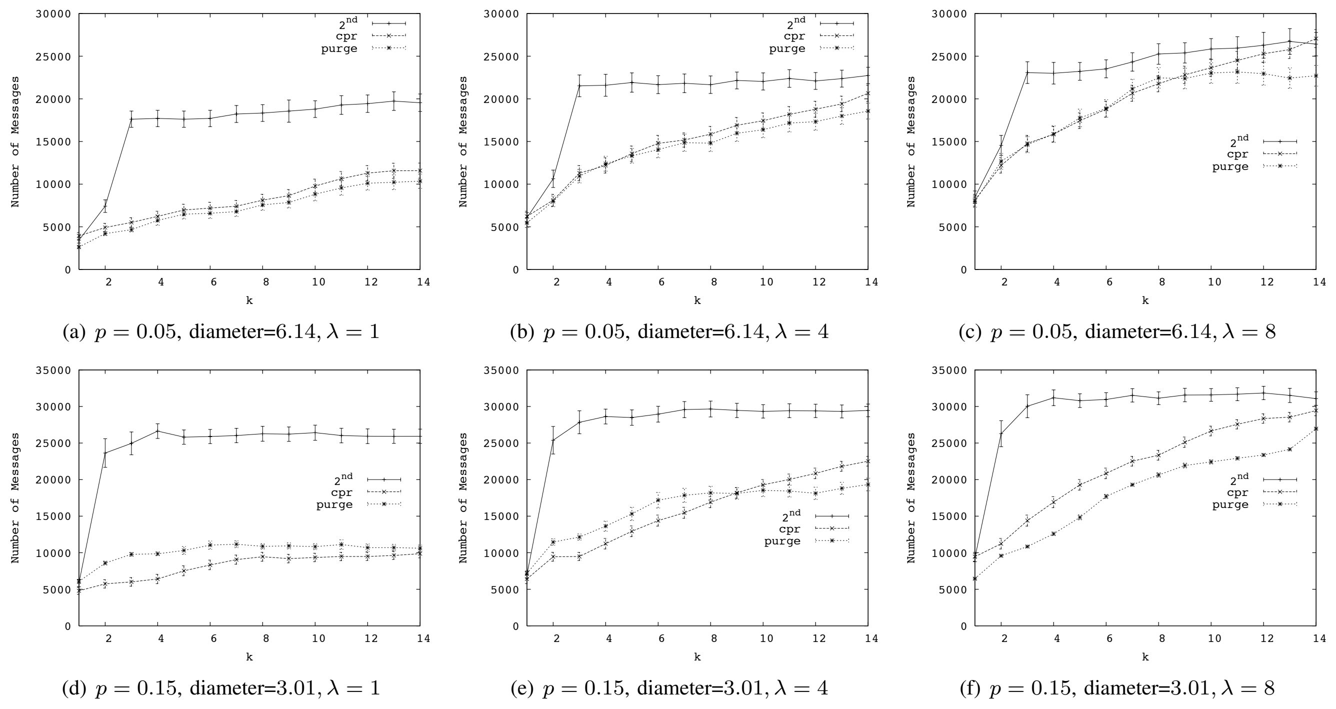


Fig. 11. Experiment 6: Message overhead for $p = \{0.05, 0.15\}$ Erdös-Rényi with link weights selected uniformly random with different λ values.

cpr has lower message overhead than purge in this case. As λ increases, cpr has higher message overhead than purge: there are more link cost changes to process and cpr must process all such link cost changes, while purge processes some link cost changes during the interval $[t', t_b]$ as part of normal distance vector execution.

2) Experiment 7 - Poison Reverse and Link Cost Changes: In this experiment, we apply poison reverse to each algorithm and repeat Experiment 6. Because purge's diffusing computations only eliminate routing loops corresponding to $\overrightarrow{bad}$ state, purge is vulnerable to routing loops stemming from link cost changes. Thus, contrary to Experiment 5, poison reverse improves purge performance. The results are shown in Figure 12. Each algorithm using poison reverse has label "algorithm-name" + pr). Results for different p values yield the same trends.

All three algorithms using poison reverse show remarkable performance gains. As confirmed by our profiling numbers, the improvements are significant because routing loops are more pervasive when link costs change. Accordingly, the poison reverse optimization yields greater benefits as λ increases.

As in Experiment 5, we believe that for $\overrightarrow{bad}$ state *only*, purge + pr removes routing loops larger than 2 while 2^{nd} best + pr does not. For this reason, we believe that purge + pr performs better than 2^{nd} best + pr. We are currently investigating this claim. cpr + pr has the lowest message complexity. In this experiment, the benefits of rolling back to a global snapshot taken before $\overline{v}$ was compromised outweigh the message overhead required to update stale state pertaining to link cost changes that occurred during $[t', t_b]$. As λ increases, the performance gap decreases because cpr + pr must process all link cost changes that occurred in $[t', t_b]$

while 2nd `best` + `pr` and `purge` + `pr` process some link cost change events during $[t', t_b]$ as part of normal distance vector execution.

However, `cpr` + `pr` only achieves such strong results by making two optimistic assumptions: we assume perfectly synchronized clocks and checkpointing occurs at each timestep. In the next experiment we relax the checkpointing assumption.

3) Experiment 8 - Vary Checkpoint Frequency: In this experiment we study the trade-off between message overhead and storage overhead for `cpr`. To this end, we vary the frequency at which `cpr` checkpoints and fix the interval $[t', t_b]$. Otherwise, our experimental setup is the same as Experiment 6.

Figure 13 shows the results for an Erdös-Rényi graph with link weights selected uniformly at random between $[1, n]$, $n = 100$, $p = .05$, $\lambda = \{1, 4, 8\}$ and $k = 2$. We plot message overhead against the number of timesteps `cpr` must rollback, z. `cpr`'s message overhead increases with larger z because as z increases there are more link cost change events to process. 2nd `best` and `purge` have constant message overhead because they operate independent of z.

We conclude that as the frequency of `cpr` snapshots decreases, `cpr` incurs higher message overhead. Therefore, when choosing the frequency of checkpoints, the trade-off between storage and message overhead must be carefully considered.

C. Summary

Our results show `cpr` using poison reverse yields the lowest message and time overhead in all scenarios. `cpr` benefits from removing false state with a single diffusing computation. Also, applying poison reverse significantly reduces `cpr` message complexity by eliminating pairwise routing loops resulting from link cost changes. However, `cpr` has storage overhead, requires loosely synchronized clocks, and requires the time $\overline{v}$ was compromised.

2nd `best`'s performance is determined by the count-to-∞ problem. In the case of Erdös-Rényi graphs with fixed unit link weights, the count-to-∞ problem was minimal, helping 2nd `best` perform better than `purge`. For all other topologies, poison reverse significantly improves 2nd `best` performance because routing loops are pervasive. Still, 2nd `best` using poison reverse is not as efficient as `cpr` and `purge` using poison reverse.

In cases where link costs change, we found that `purge` using poison reverse is only slightly worse than `cpr` + `pr`. Unlike `cpr`, `purge` makes use of computations that follow the injection of false state, that do not depend on false routing state. Because `purge` does not make the assumptions that `cpr` requires, `purge` using poison reverse is a suitable alternative for topologies with link cost changes.

Finally, we found that an additional challenge with `cpr` is setting the parameter which determines checkpoint frequency. Frequent checkpointing yields lower message and time overhead at the cost of more storage overhead. Ultimately, application-specific factors must be considered when setting this parameter.

VII. Related Work

There is a rich body of research in securing routing protocols [12], [24], [27]. However, preventative measures sometimes fail, requiring automated techniques (like ours) to provide recovery.

Previous approaches to recovery from router faults [21], [26] focus on allowing a router to continue forwarding packets while new routes are computed. We focus on a different problem - recomputing new paths following the detection of a malicious node that may have injected false routing state into the network.

Our problem is similar to that of recovering from malicious but committed database transactions. Liu [4] and Ammann [17] develop algorithms to restore a database to a valid state after a malicious transaction has been identified. `purge`'s algorithm to globally invalidate false state can be interpreted as a distributed implementation of the dependency graph approach in [17].

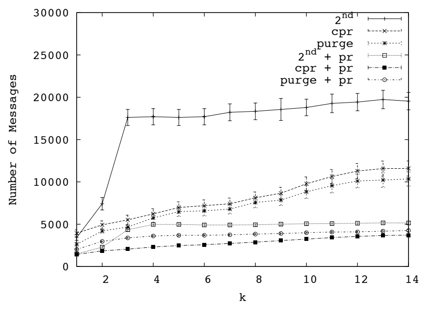
Database crash recovery [20] and message passing systems [7] both use snapshots to restore the system in the event of a failure. In both problem domains, the snapshot algorithms are careful to ensure snapshots are globally consistent. In our setting, consistent global snapshots are not required for `cpr`, since distance vector routing only requires that all initial distance estimates be non-negative.

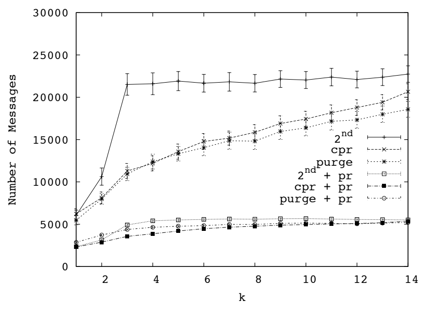
Garcia-Lunes-Aceves's DUAL algorithm [9] uses diffusing computations to coordinate least cost updates in order to prevent routing loops. In our case, `cpr` and the preprocessing procedure (Section III-A) use diffusing computations for purposes other than updating least costs (e.g., rollback to a checkpoint in the case of `cpr` and remove $\overline{v}$ as a destination during preprocessing). Like DUAL, the purpose of `purge`'s diffusing computations is to prevent routing loops. However, `purge`'s diffusing computations do not verify that new least costs preserve loop free routing (as with DUAL) but instead globally invalidate false routing state.

Jefferson [13] proposes a solution to synchronize distributed systems called Time Warp. Time Warp is a form of optimistic concurrency control and, as such, occasionally requires rolling back to a checkpoint. Time Warp does so by "unsending" each message sent after the time the checkpoint was taken. With our `cpr` algorithm, a node does not need to explicitly "unsend" messages after rolling back. Instead, each node sends its $\overrightarrow{min}$ taken at the time of the snapshot, which implicitly undoes the effects of any messages sent after the snapshot timestamp.

VIII. Conclusions and Future Work

In this paper, we developed methods for recovery in scenarios where malicious nodes inject false state into a distributed system. We studied an instance of this problem in distance vector routing. We presented and evaluated – through a theoretical analysis and simulation – three new algorithms for recovery in such scenarios. In the case of topologies with changing link costs, we found that poison reverse yields dramatic reductions in message complexity for all three algorithms. Among our


(a) $p = 0.05$, $\lambda = 1$


(b) $p = 0.05$, $\lambda = 4$

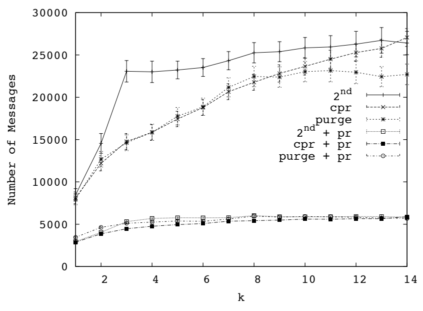
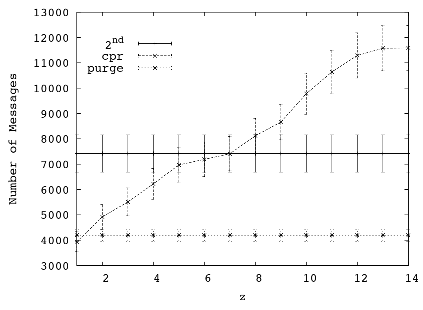
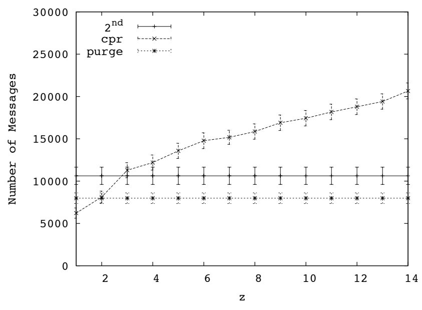

(c) $p = 0.05$, $\lambda = 8$

Fig. 12. Plots for Experiment 7. Each figure shows message overhead for Erdös-Rényi graphs with link weights selected uniformly at random, $p = 0.05$, average diameter is 6.14, and $\lambda = \{1, 4, 8\}$. The curves for 2$^{\text{nd}}$ `best + pr`, `purge + pr`, and `cpr + pr` refer to each algorithm using poison reverse, respectively.


(a) $p = 0.05$, $k = 2, \lambda = 1$


(b) $p = 0.05$, $k = 2, \lambda = 4$

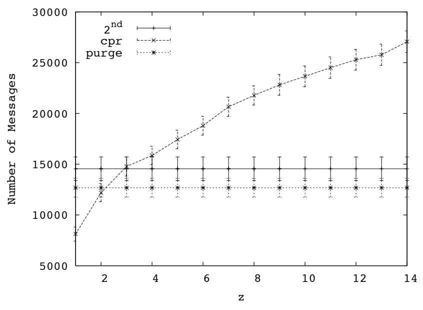

(c) $p = 0.05$, $k = 2, \lambda = 8$

Fig. 13. Experiment 8: message overhead for $p = 0.05$ Erdös-Rényi with link weights selected uniformly random with different λ values. z refers to the number of timesteps `cpr` must rollback. Note the y-axis have different scales.

three algorithms, our results showed that `cpr` – a checkpoint-rollback based algorithm – using poison reverse yields the lowest message and time overhead in all scenarios. However, `cpr` has storage overhead and requires loosely synchronized clocks. `purge` does not have these restrictions and we showed that `purge` using poison reverse is only slightly worse than `cpr` with poison reverse. Unlike `cpr`, `purge` has no stale state to update because `purge` does not use checkpoints and rollbacks.

As future work, we are interested in finding the worst possible false state a compromised node can inject (e.g., state that maximizes the effect of the count-to-∞ problem).

IX. Acknowledgments

The authors greatly appreciate discussions with Dr. Brian DeCleene of BAE Systems, who initially suggested this problem area.

References

[1] Google Embarrassed and Apologetic After Crash. http://www.computerweekly.com/Articles/2009/05/15/236060/google-embarrassed-and-apologetic-after-crash.htm.

[2] GT-ITM. http://www.cc.gatech.edu/projects/gtitm/.

[3] Rocketfuel. http://www.cs.washington.edu/research/networking/rocketfuel/maps/weights/weights-dist.tar.gz.

[4] P. Ammann, S. Jajodia, and Peng Liu. Recovery from Malicious Transactions. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1167–1185, 2002.

[5] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[6] E. Dijkstra and C. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, (11), 1980.

[7] K. El-Arini and K. Killourhy. Bayesian Detection of Router Configuration Anomalies. In *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 221–222, New York, NY, USA, 2005. ACM.

[8] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *2nd Symp. on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[9] J. J. Garcia-Lunes-Aceves. Loop-free Routing using Diffusing Computations. *IEEE/ACM Trans. Netw.*, 1(1):130–141, 1993.

[10] D. Gyllstrom, S. Vasudevan, J. Kurose, and G. Miklau. Recovery from False State in Distributed Routing Algorithms. Technical Report UM-CS-2010-017.

[11] O. Heckmann, M. Piringer, J. Schmitt, and R. Steinmetz. On Realistic Network Topologies for Simulation. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 28–32, New York, NY, USA, 2003. ACM.

[12] YC Hu, D.B. Johnson, and A. Perrig. SEAD: Secure Efficient Distance Vector Routing for Mobile Wireless Ad Hoc Networks. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 3–13, 2002.

[13] D. Jefferson. Virtual Time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[14] C. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Trans. on Knowl. and Data Eng.*, 3(4):461–473, 1991.

[15] Marjory J Johnson. Analysis of routing table update acitivity after resource recovery in a distributed computer network. pages 96–102, Honolulu, HI, 1984.

[16] Marjory J Johnson. Updating routing tables after resource failure in a distributed computer network. *Networks*, 14:379–391, 1984.

[17] P. Liu, P. Ammann, and S. Jajodia. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[18] D. Lomet, R. Barga, M. Mokbel, and G. Shegalov. Transaction Time Support Inside a Database Engine. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 35, Washington, DC, USA, 2006. IEEE Computer Society.

[19] V. Mittal and G. Vigna. Sensor-Based Intrusion Detection for Intra-domain Distance-vector Routing. In *CCS '02: Proceedings of the 9th ACM Conf on Comp. and Communications Security*, pages 127–137, New York, NY, USA, 2002. ACM.

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[21] J. Moy. Hitless OSPF Restart. In *Work in progress, Internet Draft*, 2001.

[22] R. Neumnann. Internet routing black hole. *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems*, 19(12), May 1997.

[23] V. Padmanabhan and D. Simon. Secure Traceroute to Detect Faulty or Malicious Routing. *SIGCOMM Comput. Commun. Rev.*, 33(1):77–82, 2003.

[24] D. Pei, D. Massey, and L. Zhang. Detection of Invalid Routing Announcements in RIP Protocol. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 3, pages 1450–1455 vol.3, Dec. 2003.

[25] K. School and D. Westhoff. Context Aware Detection of Selfish Nodes in DSR based Ad-hoc Networks. In *Proc. of IEEE GLOBECOM*, pages 178–182, 2002.

[26] A. Shaikh, R. Dube, and A. Varma. Avoiding Instability During Graceful Shutdown of OSPF. Technical report, In Proc. IEEE INFOCOM, 2002.

[27] B. Smith, S. Murthy, and J.J. Garcia-Luna-Aceves. Securing Distance-vector Routing Protocols. *Network and Distributed System Security, Symposium on*, 0:85, 1997.

X. Appendix

Notation. Let msg refer to a message sent during `purge`'s diffusing computation (to globally remove false routing state). msg includes:

1) a field, src, which contains the node ID of the sending node
2) a vector, $\overrightarrow{dests}$, of all destinations that include $\overline{v}$ as an intermediary node.

Let Δ refer to the maximum clock skew for `cpr`. All other notation is specified in Table I.

Algorithm 1 2

```
1:  $flag \leftarrow$ FALSE
2:  set all path costs to $\overline{v}$ to $\infty$
3:  for each destination $d$ do
4:      if $\overline{v}$ is first-hop router in least cost path to $d$ then
5:          $c \leftarrow$ least cost to $d$ using a path which does not use $\overline{v}$ as first-hop router
6:          update $\overrightarrow{min_i}$ and $dmatrix_i$ with $c$
7:          $flag \leftarrow$ TRUE
8:      end if
9:  end for
10: if $flag =$ TRUE then
11:     send $\overrightarrow{min_i}$ to each $j \in adj(i)$ where $j \neq \overline{v}$
12: end if
```

Algorithm 2 `purge`'s diffusing computation run at each $i \in adj(\overline{v})$

```
1:  set all path costs to $\overline{v}$ to $\infty$
2:  $S \leftarrow \emptyset$
3:  for each destination $d$ do
4:      if $\overline{v}$ is first-hop router in least cost path to $d$ then
5:          $\overrightarrow{min_i}[d] \leftarrow \infty$
6:          $S \leftarrow S \cup \{d\}$
7:      end if
8:  end for
9:  if $S \neq \emptyset$ then
10:     send $S$ to each $j \in adj(i)$ where $j \neq \overline{v}$
11: end if
```

Algorithm 3 `purge`'s diffusing computation run at each $i \notin adj(\overline{v})$

Input: msg containing src, $\overrightarrow{dests}$ fields.

```
1:  $S \leftarrow \emptyset$
2:  for each $d \in msg.\overrightarrow{dests}$ do
3:      if $msg.src$ is next-hop router in least cost path to $d$ then
4:          $\overrightarrow{min_i}[d] \leftarrow \infty$
5:          $S \leftarrow S \cup \{d\}$
6:      end if
7:  end for
8:  if $S \neq \emptyset$ then
9:      send $S$ to spanning tree children
10: else
11:     send $ACK$ to $msg.src$
12: end if
```

Algorithm 4 `cpr` rollback

```
1:  if already rolled back then
2:      send $ACK$ to spanning tree parent node
3:  end if
4:  $\hat{t} \leftarrow -\infty$
5:  for each snapshot, $S$, do
6:      $t'' \leftarrow S.timestamp$
7:      if $t'' < (t' - \Delta)$ and $t'' > \hat{t}$ then
8:          $\hat{t} \leftarrow t''$
9:      end if
10: end for
11: rollback to snapshot taken at $\hat{t}$
12: if not spanning tree leaf node then
13:     send rollback request to spanning tree children
14: else
15:     send $ACK$ to spanning tree parent node
16: end if
```

Algorithm 5 `cpr` "steps after rollback" run at each $i \in adj(\overline{v})$

```
1: $flag \leftarrow$ FALSE
2: for each destination $d$ do
3:     if $\overrightarrow{min_i}[d] = \infty$ then
4:         find least cost to $d$ in $dmatrix_i$ and set in $\overrightarrow{min_i}$
5:         $flag \leftarrow$ TRUE
6:     end if
7: end for
8: if $flag =$ TRUE or adjacent link weight changed during $[t', t]$ then
9:     send $\overrightarrow{min_i}$ to each $j \in adj(i)$ where $j \neq \overline{v}$
10: end if
```