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. For example, consider distance vector routing. If a compromised node incorrectly claims a distance of 0 to all destinations, this false state would likely spread throughout the network, infecting shortest paths network-wide. In this paper, we consider the problem of removing such false state.

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 [19]. 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. We make the following contributions:

- We design, develop, and evaluate three different approaches – $2^{\texttt{nd}}$ `best`, `purge`, and `cpr` – for correctly recovering from the injection of false routing state (Section II). $2^{\texttt{nd}}$ `best` performs localized state invalidation, followed by network-wide recovery using the traditional distance vector algorithm (Section II-B2). 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 (Section II-B3). Then, traditional distance vector routing is used to recompute distance vectors. `cpr` uses local snapshots and a rollback mechanism to implement recovery (Section II-B4).
- We prove the correctness of each algorithm for scenarios of single and multiple compromised nodes (Section IV). A recovery algorithm is correct if the routing tables for 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.
- We derive communication complexity bounds for each algorithm over a synchronous communication model (Section V). We find all three algorithms are bounded above by $O(mnd)$ – where d is the diamter, n is the number of nodes, and m the maximum out-degree of any node – in scenarios where link costs remain fixed. In scenarios where link costs can change, `cpr` incurs additional overhead (not experieced by $2^{\texttt{nd}}$ `best` and `purge`) because `cpr` must update stale state after rolling back. This leads to an additional term for `cpr` in its $O(mnd)$ upper bound.
- Using simulations, we evaluate the efficiency of each algorithm in terms of message overhead and convergence time in scenarios with single and multiple compromised nodes. (Section VI). Our simulations show that `cpr` using poison reverse outperforms $2^{\texttt{nd}}$ `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.

- We show through simulations that `purge` using poison reverse yields performance close to `cpr` with poison reverse in scenarios where link costs can change (Section VI-B). `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 (Section **??** and VI-B2).

Recovery from false routing state is closely related to the problem of recovering from malicious transactions [2], [15] in distributed databases. Our problem is also similar to that of rollback in optimistic parallel simulation [12]. 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 III, 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.

II. Paper Intuition

A. The Problem

We address the problem of recovering from false routing state in distance vector routing. We assume nodes are compromised such that they declare false routes to all other nodes in the network. [1] This false state spreads through the network through the normal execution of distance vector, leading to erroneous least cost paths. We assume that the identity of the compromised node is provided by a different algorithm [5], [6], [16], [20], [22], and thus do not consider this problem in this paper. Instead, we focus on recovering after receiving notification that a node is compromised. The goal is for all nodes to recover correctly: each node should remove the compromised node as a destination and find new least cost distances that do not use a compromised node. [2]

Consider the example graph, G, in Figure 1. Figure 1(a) shows G before $\overline{v}$ is compromised. [3] At this point, all least costs are correctly computed. [4] Now, let $\overline{v}$ be compromised such that it falsely declares a cost of 1 to every other node. Consistent with "good news travels fast", these false paths spread through the network, leading to erroneous least cost paths. For example, Figure 1(b) depicts the state of G after $\overline{v}$'s false routing state has propagated throughout the network. i routes via $\overline{v}$ to reach nodes l and d. j uses i to reach all nodes except l, which implies j transitively uses $\overline{v}$ to reach these destinations (e.g., j uses the path $j-i-\overline{v}$ then the false path from $\overline{v}$ to the destination). Upon receiving notification that $\overline{v}$ is compromised, the goal of the recovery is to reach the state depicted in Figure 1(c): $\overline{v}$ is removed as a destination and no least cost uses $\overline{v}$ as an intermediate node.

B. Our Solutions

In this section we propose three new recovery algorithms: 2^{nd} `best`, `purge`, and `cpr`. The input and output of each algorithm is the same. The input for each algorithm is an undirected graph, correctly computed least cost paths to all nodes, and the identity of the compromised node. The output of each algorithm is a new undirected graph with the compromised node removed and new least cost paths that route around the compromised node.

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

1) Preprocessing Procedure: All three recovery algorithms share a common preprocessing procedure. The procedure removes the compromised node as a destination and finds the node IDs in each connected component. This is implemented using diffusing computations [4] initiated at each neighbor of the compromised node. 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.

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 own node 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.

2) 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 II-B1, each neighbor of the compromised node locally invalidates state by selecting the least cost pre-existing path

[1] For simplicity, in this section we present our recovery algorithms in the case of a single compromised node. The necessary extensions to handle multiple compromised nodes are addressed in our Technical Report [9].

[2] 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.

[3] For convenience, we refer to $\overline{v}$ as the generic compromised node in the remainder of the document.

[4] Node i and j's routing table are shown to the right of G. The least costs are underlined.

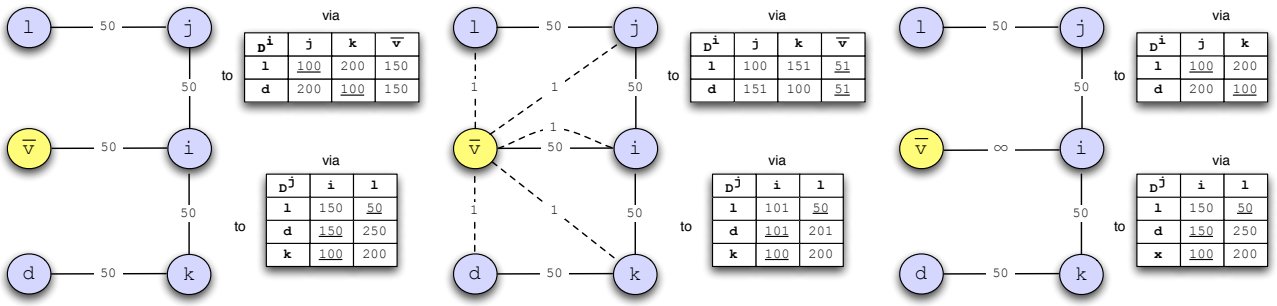


Fig. 1. Three snapshots of a graph, G, where $\overline{v}$ is the compromised node: (a) G before $\overline{v}$ is compromised, (b) G after false state has finished propagating but before recovery has started, and (c) G after recovery. The dashed lines in (b) mark false paths. Portions of node i's and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

that does not use the compromised node as the first hop. The resulting distance vectors trigger the execution of traditional distance vector, which removes the remaining false state.

We trace the execution of 2^{nd} `best` using the example in Figure 1. After the preprocessing completes, i selects a new neighbor to route through to reach l and d by finding its new smallest distance in its local routing table 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, i sends its new least cost vector to its neighbors, triggering the execution of traditional distance vector.

3) 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. [5] The diffusing computation is initiated at the neighbors of the compromised node ($\overline{v}$) because only these nodes are aware if $\overline{v}$ is used an intermediary node along a given path. 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. Finally, `purge` uses distance vector to recompute least cost paths invalidated by the diffusing computations.

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 ∞. To do so, i uses its local routing table to find its new least cost to l and d. This triggers the execution of distance vector, which recomputes least cost paths invalidated by the diffusing computations. We prove that `purge` is loop-free in Corollary 19.

`purge` is very similar to Garcia-Lunes-Aceves's DUAL algorithm [8]. [6] DUAL uses diffusing computations to coordinate least cost computations after a node (or link) fails. [7] The diffusing computations find a next-hop node – called a feasible successor – which ensures loop freedom. Once a node finds a feasible successor and has received replies (corresponding to a diffusing computation) from all child nodes, the node operates according to distance vector: if a new least cost is selected, an update is sent to the node's neighbors. [8] As with `purge`, this ensures that only valid paths are shared.

A subtle difference between DUAL and `purge` is that `purge` initiates distance vector computations – to recompute valid paths to destinations invalidated by the diffusing computations – from the neighbors of the compromised node, while DUAL starts distance vector computations once a node finds a feasible successor and has received replies from all child nodes. With DUAL, this occurs close to the leaves of the diffusing computation spanning trees.

4) The cpr Algorithm: `cpr` adds a time dimension to each node's routing table, which `cpr` uses to locally archive a complete history of values. The routing table snapshots are taken either at a given frequency or after some number of distance value changes (e.g., each time a distance value changes). Once

[5] Because diffusing computations preserve the decentralized nature of distance vector, `purge` is a decentralized algorithm.

[6] `purge` was designed prior to the authors' efforts to consider the DUAL algorithm as a solution to the false-state recovery problem.

[7] Although DUAL does not explicitly consider the false routing state problem, with some minor changes DUAL can be adapted to recover in such scenarios. DUAL assumes nodes locally detect a failing node. In contrast, our algorithms assume detection is handled by an outside algorithm. Thus, DUAL must be modified such that all nodes accept input from a detection algorithm.

[8] DUAL does not explicitly invalidate routing state from a failed node but DUAL's diffusing computations to find a feasible successor have a similar effect.

nodes are notified that a node is compromised, the nodes roll back to a snapshot taken before $\overline{v}$ was compromised. Then, cpr removes $\overline{v}$ as destination and runs distance vector to update stale distance values resulting from link cost changes.

In the example from Figure 1, cpr has a snapshot that reflects the system state shown in Figure 1(a). After $\overline{v}$ is compromised, cpr rolls back to said snapshot. Finally, cpr runs standard distance vector to update the stale state (e.g., $(i,\overline{v}) = 50$ in the snapshot rather than ∞).

C. 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, $\overline{V}$, the changes to each algortihm are straighforward. For each $\overline{v} \in \overline{V}$, all $adj(\overline{v})$ are notified that $\overline{v}$ was compromised. The the diffusing computations described in Section II-B1 are initiated at the neighbor nodes of each node in $\overline{V}$. [9] From this point, no additional changes are needed.

More changes are required to handle the case where additional nodes are compromised during the execution of a recovery algorithm. Consider the case where $\overline{v}_2$ is compromised during the recovery initiated by the comrpomise of $\overline{v}_1$. Any node receiving a distance vector message with an updated distance value to destination $\overline{v}_2$, ignores this value. [10] Without this change it is possible that the recovery algorithm will not terminate. In our example, 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 $\overline{v}_1$ and $\overline{v}_2$ to ∞. If the first distance vector execution triggered by $\overline{v}_1$'s compromise is not modified to ignore distance values to $\overline{v}_2$, the distance vector computation would never complete because the least cost to $\overline{v}_2$ is ∞.

III. Formal Problem Statement and Notation

We consider distance vector routing [3], [7] over arbitrary network topologies. We model a network as an undirected graph, $G = (V, E)$, with a link weight function $w : E \to \mathbb{N}$. [11] 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 ($dmatrix_v$).

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

[10]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.

[11]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.

We make the following assumptions about the distance vector computation. All 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 are non-negative and do not change. G is finite and connected. Finally, we assume reliable communication.

We assume that the identity of the compromised nodes – which we refer to as $\overline{V}$ – are provided by a different algorithm [5], [6], [16], [20], [22]. Specifically, we assume that at time t_b, this algorithm detects all compromised nodes and notifies the neighbors of each compromised node. Let t' be the time the first node was compromised.

For each algorithm, the goal is for all nodes to recover correctly: all nodes should remove all 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 nodes, all nodes need only compute new least cost distances to all other nodes within their connected component. With one exception, the input and output of each algorithm is the same. [12]

- **Input:** Undirected graph, $G = (V, E)$, with weight function $w : E \to \mathbb{N}$. $\forall v \in V$, $\overrightarrow{min}_v$ and $dmatrix_v$ are computed (using distance vector). Also, each $v \in adj(\overline{v})$ is notified that $\overline{v}$ was compromised.
- **Output:** $G' = (V', E')$, where $V' = V - \overline{V}$, $E' = E - \{(\overline{v}, v_i) \mid \overline{v} \in \overline{V} \wedge v_i \in adj(\overline{v})\}$.

For convenience, $|V| = n$ and the diameter of G' is d. Let $\max_{i \in V'}(|adj(i)|) = m$.

For an arbitrary $\overline{v} \in \overline{V}$ we use the following notation. $\overrightarrow{old}$ refer to $\overrightarrow{min}_{\overline{v}}$ before $\overline{v}$ was compromised. $\overrightarrow{bad}$ denotes $\overrightarrow{min}_{\overline{v}}$ 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.

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 t_b and $\delta_s(i, j)$ is the cost of this path; $p_u(i, j)$ is i's least cost path to j at time $t \in [t_b, t^*]$ and $\delta_u(i, j)$ the cost of this path [13]; 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'.

For each algorithm, let t^* mark the time when the recovery algorithm completes. Let $\hat{t}$ be 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 $\overline{v}$

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

[13]$p_u(i, j)$ and $\delta_u(i, j)$ can change during $[t_b, t^*]$.

Notation	Meaning
$\overline{V}$	set of compromised nodes
G	undirected graph (V, E), with weight function $w : E \rightarrow \mathbb{N}$
G'	undirected graph (V', E'), where $V' = V - \overline{V}$, $E' = E - \{(\overline{v}, v_i) \mid \overline{v} \in \overline{V} \wedge v_i \in adj(\overline{v})\}$
$adj(v)$	nodes adjacent to v in G'
n	$\lvert V \rvert$
d	diameter of G'
m	$\max_{i \in V'}(\lvert adj(i) \rvert) = m$
$\ell(i, j)$	minimum number of links between nodes i and j in G'
$dmatrix_i$	node i' distance matrix
$\overrightarrow{min_i}$	node i's the least cost vector
$\overrightarrow{bad}$	compromised node's least cost vector at $t \geq t'$
$\overrightarrow{old}$	compromised node's least cost vector at $t < t'$
$p_t(i, j)$	i's actual least cost path to j at time t.
$\delta_t(i, j)$	i's least cost between nodes to j at time t
$p_s(i, j)$	i's least cost path to j at time t_b
$\delta_s(i, j)$	i's least cost to j at time t_b
$p_u(i, j)$	i's least cost path to j at time $t \in [t_b, t^*]$
$\delta_u(i, j)$	i's least cost to j at time $t \in [t_b, t^*]$
$p_f(i, j)$	i's least cost path to j at t^*
$\delta_f(i, j)$	i's least cost to j at t^*
t_b	time the compromised node is detected
t'	time the compromised node was compromised
t^*	time when recovery algorithm completes
$\hat{t}$	time all diffusing computations complete

TABLE I
NOTATION TABLE

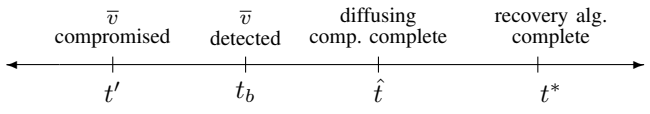


Fig. 2. Time line with important timesteps labeled.

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.

Table I summarizes the notation used in this document and all important timesteps are shown in Figure 2.

IV. CORRECTNESS OF RECOVERY ALGORITHMS

Our correctness proofs consider the general case where multiple nodes are compromised. We assume the computation model described in Section III.

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 [3] 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. `2nd 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 $dmatrix_v$ values are non-negative for all $v \in V'$, by Theorem 1 we conclude `2nd best` is correct if no additional node(s) are compromised during $[t', t^*]$. ■

Corollary 3. `2nd 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 `2nd best` is correct in the case where a set of nodes, $\overline{V}_2$, are compromised concurrent with a running execution of `2nd 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 II-C. 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 `2nd best` is correct.

Since we have proved `2nd best` is correct when multiple nodes are simultaneously compromised and when nodes are compromised concurrent with any `2nd best` execution, we conclude that `2nd 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 II-B1, 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 II-C 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 derive communication complexity bounds for each recovery algorithm. First, we consider graphs where link costs remain fixed (Section V-A – Section V-D). Then, we derive bounds where link costs can change (Section V-E).

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).

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., $\forall j \in V', \delta_s(\overline{v}, j) = 1$).

A. Diffusing Computation Analysis

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

Lemma 8. *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 9. *The diffusing computations for* 2^{nd} `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 8, each diffusing computation has $O(E)$ communication complexity, yielding $O(mE)$ communication complexity. ■

Definition 2. $C(i,j) = \delta_f(i,j) - \delta_{\hat{t}}(i,j)$.

In other words, $C(i,j)$ is the magnitude of change in $\delta(i,j)$ between time the diffusing computations for each algorithm complete and when the final correct least costs are computed over G'.

B. 2^{nd} `best` Analysis

Johnson [14] 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. [14] From this point, DV behaves exactly like 2^{nd} `best`. [15] 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 5 in our technical report [9], 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 characterization of 2^{nd} `best` recovyer, Theorem 10 derives upper and lower

[14] 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.

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

bounds for 2nd 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 10. *After $\hat{t}$, 2nd 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 [14] 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, 2nd 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 [14]. Theorem 2 gives us an 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. ∎

Corollary 11. *2nd best has $O(mnd)$ communication complexity.*

Proof: From Lemma 9, 2nd best's diffusing computations have $O(mE)$ communication complexity. After the difussing computations complete, 2nd 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 2nd best has $O(mnd)$ communication complexity. ∎

Next, we restate Theorem 1 from Johnson [14] using our notation. Theorem 12 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 12. *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 12 translates the problem of finding the number of update messages after false node detection into one 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 by 2nd best.

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 $\overline{v}$. A node i is singular if $i \in S(p)$. Let $q(\overline{v}, i)$ be the smallest positive integer p such that $i \in S(p)$. For convenience, we refer to and $q(\overline{v}, i) = c$.

The next two theorems, Theorem 14 and 15, follow from Theorem 5 in [14] and Theorem 12. For convenience, we restate Theorem 5 from [14] here:

Theorem 13. *If a graph remains connected after a compromised node is detected, a node is singular if and only if the graph contains an odd cycle.*

Theorem 14. *If G contains no odd cycles, the number of update messages after $\hat{t}$ is described exactly by Equation 1.*

Theorem 15. *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 12 tells us that before converging on the correct distance to a destination, a, 2nd best exhaustively searches all paths from i to $\overline{v}$ and then uses $\overline{v}$'s false path to a. If G has no odd cycles, then i counts up by 2 until reaching the final correct cost to a. Node i does so by using a path that hops 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 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 $\delta_f(i, j)$.

C. 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 10, Theorem 14, and Theorem 15 apply directly to cpr, the bounds and exact message count can defer

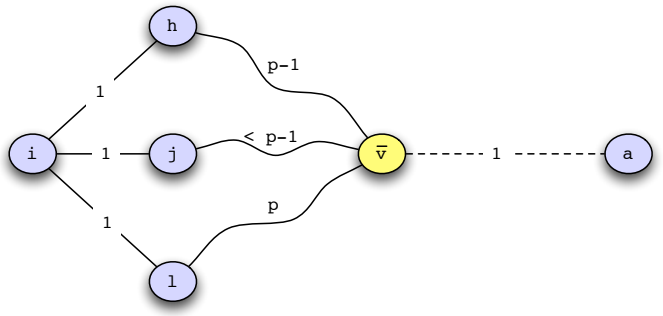


Fig. 3. The yellow node ($\overline{v}$) is the compromised node. The dotted line from $\overline{v}$ to a represents the false path.

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. [16] Thus, cpr's $C(i,j)$ values are typically smaller than those for 2nd best, resulting in lower message complexity for cpr.

D. 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 16). We establish that the first finite cost to the destination is in fact the node's final correct least cost to the destination (Theorem 18). 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 an 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.

Definition 3. *Let $B(a,t)$ be the set of nodes that have least cost ∞ to destination node a at time t.*

Definition 4. *$F(a,t)$ 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 16 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 16. *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 a result, each $i \in B(a,\hat{t})$ sends a DV message with a least of ∞ to each neighbor, [17] unless i has a neighbor node in $F(a,\hat{t})$. 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 a node i that has changed $\delta_t(i,a)$ from ∞ to a finite value must have $j \in adj(i)$ 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 (17) 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 18.

Lemma 17. *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 $\ell(b,\overline{v}) > \ell(b,a)$. Since we have assumed $b \in F(a,\hat{t})$, there must exist $b' \in adj(b)$ such that $b' \in B(a,\hat{t})$. 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 18 proves that once any $i \in V'$ changes its least cost from ∞, i changes its least cost to the final correct value.

Theorem 18. *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 $\delta_f(i,a)$.*

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 16 we proved that i does not update its least cost to a finite value until it receives a least cost from a $b \in F(a,\hat{t})$. [18] For the sake of contradiction, assume that for some $b' \in F(a,\hat{t})$, where $b' \neq b^*$, that $\delta_f(b',a)$

[16] 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., $\overrightarrow{bad}$=$\overrightarrow{old}$).

[17] 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 $\hat{t}$).

[18] 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.

reaches i before $\delta_f(b^*, a)$. [19] This implies that:

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

From Lemma 17, 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:

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

However, since we have assumed that i routes via b^* in $p_f(i, a)$, we know that:

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

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 17. 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 19. `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 18 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 20. `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 9, `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 18 and Lemma 16, 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 hops 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 $O(mnd)$. ■

E. Analysis with Link Cost Changes

In this section, we analyze each of our algorithms in the case where w link cost changes occur during $[t', t_b]$.. In our analysis, we assume that all w link cost changes finish propagating before $\overline{v}$ is detected (e.g., before t_b).

[19] From Lemma 16 we know that a finite least cost to a reaches every node in $B(a, \hat{t})$.

The analysis for `2nd best` and `purge` from Section **??** and Section **??**, 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 **??** 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 $C'(i, j) = \delta'_f(i, a) - \delta_{\hat{t}}(i, j)$. That is, $C'(i, j)$ is i's least to j if no link cost changes occurred minus i's least cost to j after the diffusing computations complete.

The communication complexity for a link cost increase is $O(n^2)$ [14] and $O(E)$ for a link cost decrease [13]. Let there be u link cost increases and $w - u$ link cost decreases. At worst, the link cost changes are processed after $\overline{v}$ recovery completes. As a result, `cpr`'s least cost counts up by $C'(i, j)$ and then `cpr` processes the link cost changes. Thus, `cpr`'s communication complexity with link cost changes is bounded above by:

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

1) 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 different performance of our three algorithms is determined by the hidden constants.

We also bounded the communication overhead of each algorithm under conditions of link cost changes. `cpr` incurred overhead not experienced by `2nd best` and `purge` because these two algorithms 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. Trends are consistent with an asynchronous implementation. For ease of exposition, we present the results from the synchronous model.

Except for Section VI-A2, we consider the case of a single compromised node. For convenience, we refer to the compromised node as $\overline{v}$, $\overrightarrow{min_{\overline{v}}}$ before t' as $\overrightarrow{old}$, and $\overrightarrow{min_{\overline{v}}}$ after t' as $\overrightarrow{bad}$.

We simulate the following scenario: [20]

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_b, some node $v \in V$ notifies all $v \in adj(\overline{v})$ that $\overline{v}$ was compromised. [21]

The message and time overhead are measured in step (4) above. The pre-computation common to all three recovery algorithms, described in Section II-B1, is not counted towards message and time overhead. We describe our simulation scenario for multiple compromised nodes in Section VI-A2.

A. Fixed Link Weight Graph Simulations

In our previous paper [10], we found `cpr` outperforms `2nd best` and `purge` because `cpr` removes false routing state with a single diffusing computation, rather that using an iterative distance vector computation (as with `2nd best` and `purge`). We showed that `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. As a result, `2nd best` performs better than `purge`. For all other topologies the count-to-∞ problem is significant for `2nd best`. Because `purge` avoids the count-to-∞ problem by globally invalidating false state before computing new least cost paths, `purge` has significantly lower message overhead than `2nd best`.

We also found in [10] that poison reverse optimization yields only modest gains for `cpr` because few routing loops occur with `cpr`. [22] In contrast, 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 the next two simulations we build on our evaluation from [10]. We compare our algorithms with Garcia-Lunes-Aceves's DUAL [8] algorithm (Simulation 1) and evaluate our recovery algorithms in the case of multiple compromised nodes (Simulation 2).

1) Simulation 1 - Comparison with DUAL: In this simulation, we compare the performance of our algorithms with Garcia-Lunes-Aceves's DUAL algorithm [8]. We focus here on the case of Erdös-Rényi graphs with link weights distributed uniformly at random. Simulations for Erdös-Rényi graphs with unit link weights and for the other scenarios described in Section VI-B yield the same trends.

[20] In Section VI-A2, we modify our simulation scenario to consider a set of compromised nodes, $\overline{V}$, instead of $\overline{v}$.

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

[22] Recall by Corollary 19, `purge` is loop-free. As such, the poison reverse optimization has no effect on `purge` message and time overhead.

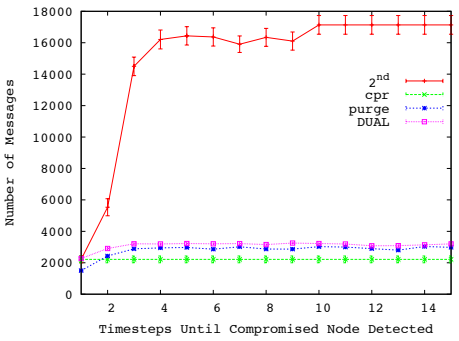


Fig. 4. Comparison with DUAL. Erdös-Rényi graphs with link weights distribution uniformly at random.

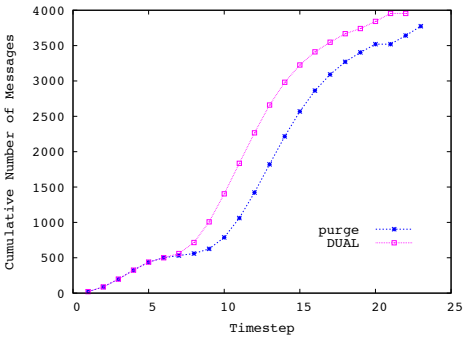


Fig. 5. Comparison with DUAL. Erdös-Rényi graphs with link weights distribution uniformly at random.

As expected, `purge` yields slightly better performance than DUAL (Figure 4). Recall from Section II-B3, the only difference between `purge` and DUAL is where distance vector computations are initiated: `purge` initiates new distance vector computations at the neighbors of the compromised node, while DUAL does so at the leaf nodes of each diffusing computation. As a result, on average more nodes initiate distance vector computations with DUAL: on average 5 nodes start distance vector computations with `purge` but with DUAL, on average (across all k values), 59 nodes initiate diffusing computations. As a result, we observe a spike in message overheard for DUAL – when compared to `purge` – after the diffusing computations complete. As evidence, consider Figure 5, which plots the average cumulative number of messages per timestep for $k = 5$ (the same trends were found for other values of k). The plot shows identical message complexity until about epoch 7, when the diffusing computation complete. We then see a sharper increase for DUAL's message complexity over the next few timesteps. After this point, DUAL and `purge` message complexity increase at roughly the same rate.

We conclude that the additional overhead in initiating distance vector computations at the leaves of the diffusing computations results in more overhead for DUAL than with `purge`'s approach of starting distance vector computations from the neighbors of the compromised node.

2) Simulation 2 - Multiple Compromised Nodes: Here we evaluate our recovery algorithms when multiple nodes are

compromised. Our setup is different from what we have used to this point: we fix $k = \infty$ and vary the number of compromised nodes. This simulates the case all compromised nodes are not detected until their false routing state has finished propagating. 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 $[t', t'+10]$ (**why** 10). The simulation is setup so that the outside algorithm identifies all m compromised node at time t_b. 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 simulate this scenario using Erdös-Rényi graphs with fixed link costs. The message overhead results are shown in Figure 6(a) for $p = 0.05$ and $n = 100$. [23] The relative performance of the three algorithms is consistent with the results from Experiment 1 from [10], in which we had a single compromised node. As in Experiment 1 from [10], 2^{nd} `best` and `cpr` have few pairwise routing loops (Figure 6(b)). In fact, there is more than an order of magnitude fewer pairwise routing when compared to the results for the same simulation scenario of m compromised nodes using Erdös-Rényi graphs with random link weights (Figure 7(b)). Few routing loops imply that 2^{nd} `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_b.

2^{nd} `best` and `purge` message overhead are nearly constant for $m \geq 8$ because at that point $\overrightarrow{bad}$ state has saturated G. Figure 6 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 6(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 7(a). Below, we explain the performance of each algorithm in detail.

Consistent with Experiment 2 and 3 from [10], 2^{nd} `best` performs poorly because of the count-to-∞ problem. Figure 7(b) shows that a significant number of pairwise routing loops occur during 2^{nd} `best` recovery. 2^{nd} `best` message overhead remains constant when $m \geq 6$ because at this point $\overrightarrow{bad}$ state has saturated the network. Figure 7(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 7(c)). More $\overrightarrow{old}$ state results in more routing loops when distance vector is run after rolling back – as shown in Figure 7(b) – causing increased message overhead.

`purge` performs the best because unlike `cpr` and 2^{nd} `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 in [10]. The results were consistent with those for Erdös-Rényi graphs with random link weights.

B. Link Weight Change Simulations

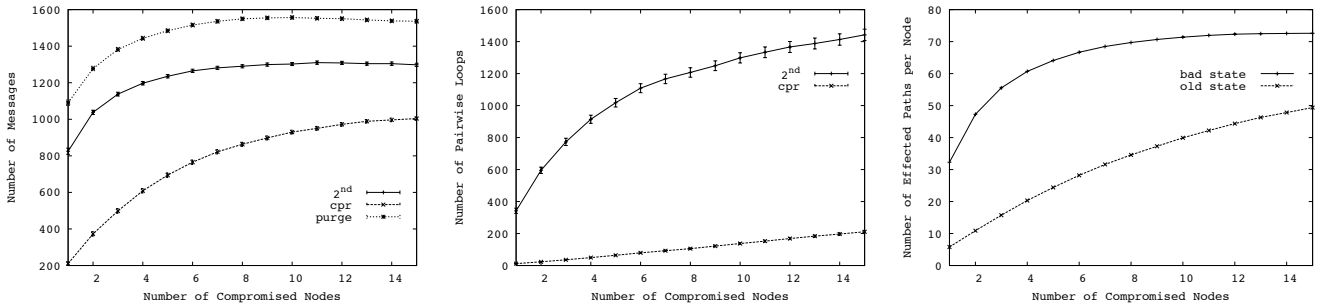
In the next three simulations we evaluate our algorithms in scenarios with graph link costs change. We introduce link cost changes between the time $\overline{v}$ is compromised and when $\overline{v}$ is discovered (e.g., during $[t', t_b]$). In particular, let there be λ link cost changes per timestep, where λ is deterministic. To create a link cost change event, we choose a link $(i, j) \in E'$ uniformly at random and set (i, j)'s link cost to be between $[1, n]$ uniformly at random.

1) Simulation 4 - Link Cost Changes: Except for λ, our experimental setup is identical to the one in Simulation 2 (**fix**).. 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 8 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.

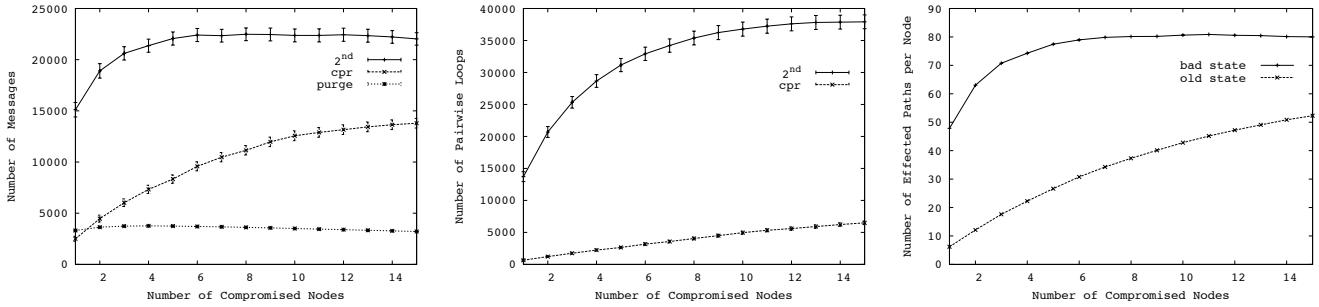
Our analysis further indicates that 2^{nd} `best` performance suffers because of the count-to-∞ problem. The gap between 2^{nd} `best` and the other algorithms shrinks as λ increases because as λ increases, link cost changes have a larger effect on message overhead.

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



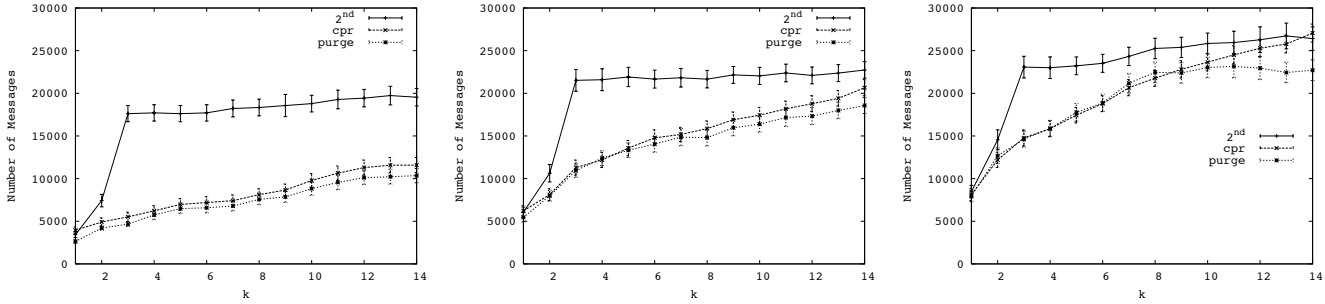
(a) Message Overhead (b) Pairwise Routing Loops (c) Number of Effected Least Cost Paths

Fig. 6. Simulation 2 - 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. 7. Simulation 2 - 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.



(a) $p = 0.05$, diameter=6.14, $\lambda = 1$ (b) $p = 0.05$, diameter=6.14, $\lambda = 4$ (c) $p = 0.05$, diameter=6.14, $\lambda = 8$

Fig. 8. Simulation 4: Message overhead for $p = 0.05$ Erdös-Rényi with link weights selected uniformly random with different λ values.

With larger p values (refer to our Technical Report [9] for the figures), 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, 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) Simulation 5 - Poison Reverse and Link Cost Changes: In this simulation, we apply poison reverse to each algorithm and repeat Simulation 4. 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 Simulation 4, poison reverse improves purge performance. The results are shown in Figure 9. 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 Simulation 4, we believe that for $\overrightarrow{bad}$ state *only*, `purge + pr` removes routing loops larger than 2 while `2nd best + pr` does not. For this reason, we believe that `purge + pr` performs better than `2nd 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 favorable results under two optimistic assumptions: we assume perfectly synchronized clocks and checkpointing occurs at each timestep. In the next simulation we relax the checkpointing assumption.

3) Simulation 6 - Vary Checkpoint Frequency: In this simulation 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 Simulation 4.

Figure 10 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

todo mention additional simulations: DUAL and multiple bad nodes. 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 as input 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 [11], [21], [24]. However, preventative measures sometimes fail, requiring automated techniques (like ours) to provide recovery.

Previous approaches to recovery from router faults [18], [23] 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. Similarities with Garcia-Lunes-Aceves's DUAL algorithm [8] are discussed in Section II-B3 and Section VI-A1.

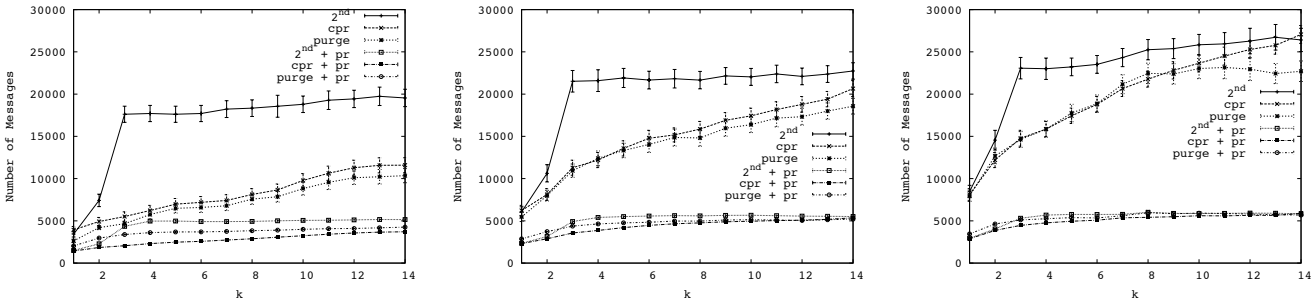
Our problem is similar to that of recovering from malicious but committed database transactions. Liu [2] and Ammann [15] develop algorithms to restore a database to a valid state after a malicious transaction has been identified. `purge`'s algorithm to globally invalidate false state can be interpreted as a distributed implementation of the dependency graph approach in [15].

Database crash recovery [17] and message passing systems [5] 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.

Jefferson [12] 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.

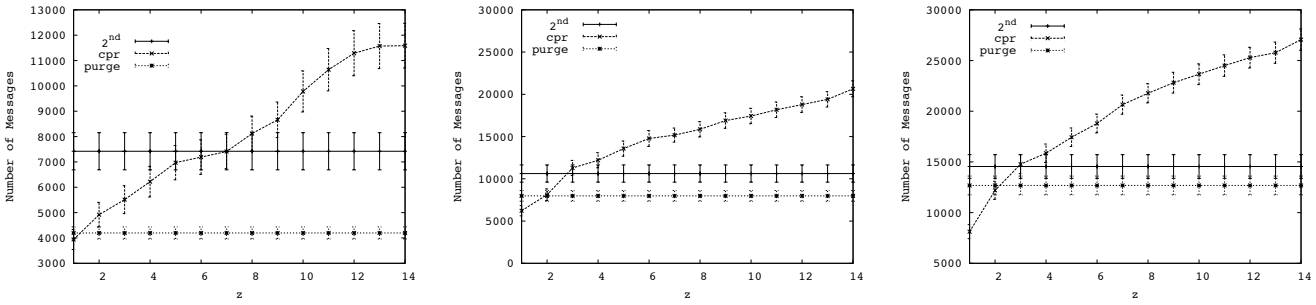
VIII. Weaknesses

Our evaluation fails to consider a lower bound – both in terms of message and time overhead – on recovery. Our theoretical and simulation study both consider the relative



(a) $p = 0.05$, $\lambda = 1$ (b) $p = 0.05$, $\lambda = 4$ (c) $p = 0.05$, $\lambda = 8$

Fig. 9. Plots for Simulation 5. 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^{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. 10. Simulation 6: 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.

performance of our recovery algorithms and the DUAL algorithm. A lower bound would yield insights into the overall performance of all three of our recovery algorithms: without a lower bound it is possible that all three of our recovery algorithms are inefficient. However, we believe this is unlikely.

Our model for compromised node behavior is simplistic (e.g., nodes falsely claim a cost of 1 to all other nodes). Clearly, this makes the detection of a compromised node easy. Because our focus is on false state recovery, rather than false state detection, we believe our simplistic model is still a meaningful context to evaluate our recovery algorithms. Furthermore, since we are unaware of any existing approach that explicity considers this problem, we feel it is appropriate to start with the simplest problem formulation that succeeds in revealing the fundamental challenges of the false-state recovery problem. We believe we have succeeded in this aim.

IX. 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 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 exploring other false state vectors. For example, finding the worst possible false state a compromised node can inject (e.g., state that maximizes the effect of the count-to-∞ problem). In addition, we would like to derive a lower bound – for both message and time complexity – for recovery.

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

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

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

[5] 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.

[6] 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.

[7] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, New Jersey, 1962.

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

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

[10] D. Gyllstrom, S. Vasudevan, J. Kurose, and G. Miklau. Efficient recovery from false state in distributed routing algorithms. In *Networking*, pages 198–212, 2010.

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

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

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

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

[21] 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.

[22] 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.

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

[24] 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.