Efficient Recovery from False State in Distributed Routing Algorithms

Daniel Gyllstrom, Sudarshan Vasudevan, Jim Kurose, and Gerome Miklau

Department of Computer Science, University of Massachusetts Amherst
140 Governors Drive Amherst, MA 01003
{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 and prove the correctness of each of these algorithms. 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.

Keywords: Routing, Security, Recovery, Checkpointing, Fault Tolerance

1 Introduction

Malicious and misconfigured nodes can degrade the performance of a distributed system by injecting incorrect state information. Such false state can then further propagate through the system either directly in its original form or indirectly, e.g., by 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, recently a routing error forced Google to redirect its traffic through Asia, causing congestion that left many Google services unreachable [1]. 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, implement, 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 has a least cost path to any destination that routes through a compromised node.

Specifically, we develop three new distributed recovery algorithms: 2nd `best`, `purge`, and `cpr`. 2nd `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 2nd `best`, traditional distance vector routing is then used to recompute distance vectors. `cpr` uses local snapshots and a rollback mechanism to implement recovery.

We prove the correctness of 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 2nd `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 has similarities to the problem of recovering from malicious transactions [12] in distributed databases. Our problem is also similar to rollback in optimistic parallel simulation [11]. We are unaware of existing solutions to the problem of recovering from false routing state. However, a related problem is that of discovering misbehaving nodes. In Section 2, 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 five sections. In Section 2 we define the problem and state our assumptions. We present our three recovery algorithms in Section 3. Section

4 describes our simulation study. We detail related work in Section 5 and finally we conclude and comment on directions for future work in Section 6.

2 Problem Formulation

We consider distance vector routing [4] over arbitrary network topologies. [1] We model a network as an undirected graph, $G = (V, E)$, with a link weight function $w : E \rightarrow \mathbb{N}$. 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 our technical report [9].

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 [6, 7, 13, 16, 18]. Specifically, we assume that at time t, 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 node as a destination and find new least costs that do not use the compromised node. If the network becomes disconnected as a result of removing the compromised node, all nodes need only compute new least costs to the nodes in their connected component. For simplicity, let $\overline{v}$ denote the compromised node, let $\overrightarrow{old}$ refer to $\overrightarrow{min_{\overline{v}}}$ before the first $\overline{v}$ 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.

3 Recovery Algorithms

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

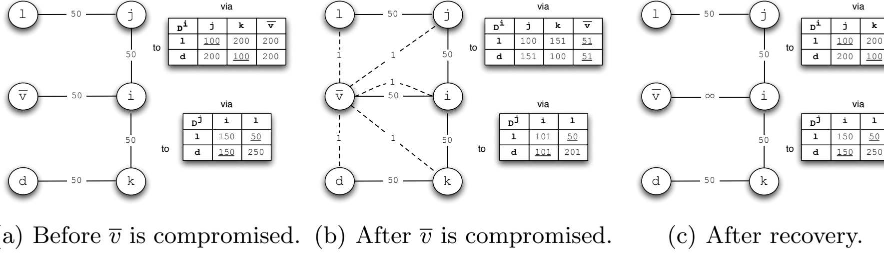
Input: Undirected graph, $G = (V, E)$, with weight function $w : E \rightarrow \mathbb{N}$. $\forall v \in V$, $\overrightarrow{min_v}$ and $dmatrix_v$ are computed (using distance vector). Also, each $v \in adj(\overline{v})$ is notified that $\overline{v}$ was compromised.

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 $\forall v \in V'$.

---

[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.

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



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

First we describe a preprocessing procedure common to all three recovery algorithms. Then we describe each recovery algorithm. Due to space constraints, the proof of correctness and pseudo code for each algorithm can be found in our technical report [9].

3.1 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 [5] 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, 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 only computes shortest paths to nodes in the vector.

3.2 The 2nd `best` Algorithm

2nd `best` invalidates state locally and then uses distance vector to implement network-wide recovery. Following the preprocessing described in Section 3.1, 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.

We trace the execution of 2$^{\text{nd}}$ `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 next-hop node 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$^{\text{nd}}$ `best` is simple and makes no synchronization assumptions. However, 2$^{\text{nd}}$ `best` is vulnerable to the count-to-∞ problem: because each node only has local information, the new shortest paths may continue to use $\overline{v}$.

3.3 The `purge` Algorithm

`purge` globally invalidates all false state using diffusing computations 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 computations are initiated at the neighbors of $\overline{v}$ and spread 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. Next, `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 from j and k, 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 to find i's new route to d (which is via k). Finally, i sends $\overrightarrow{min_i}$ to its neighbors, triggering the execution of distance vector to recompute the remaining distance vectors.

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.

3.4 The cpr Algorithm

`cpr`[3] is our third and final recovery algorithm. Unlike 2nd `best` and `purge`, `cpr` requires that clocks across different nodes be loosely synchronized. We assume a maximum clock offset between any two nodes. For ease of explanation, we describe `cpr` as if clocks 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, $i \in V$, `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 $\overline{v}$, $\overrightarrow{old}$, and update stale distance values resulting from link cost changes. We describe each algorithm step in detail below.

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.

Our distributed archive algorithm is quite simple. Each node can archive 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. [5]

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.

Step 3: Steps after rollback. After Step 2, the algorithm in Section 3.1 is executed. When the diffusing computations complete, there are two issues to address. First, nodes may be using $\overrightarrow{old}$. Second, nodes may have stale state as a result of link cost changes that occurred during $[t', t]$ and consequently are not reflected in the snapshot. To resolve these issues, each $i \in adj(\overline{v})$ 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.

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

[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.

[5] A globally consistent snapshot is not required for correctness [9].

$(i,\overline{v}) = 50$ rather than ∞. Step 3 of `cpr` makes this change. Because no nodes use $\overrightarrow{old}$, no other changes take place.

In summary, 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 $[t', t]$. 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.

4 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 evaluate recovery after a single compromised node has distributed false routing state. An evaluation of our algorithms in the case of multiple compromised nodes can be found in our technical report [9].

We build a custom simulator with a synchronous communication model: 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). 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:

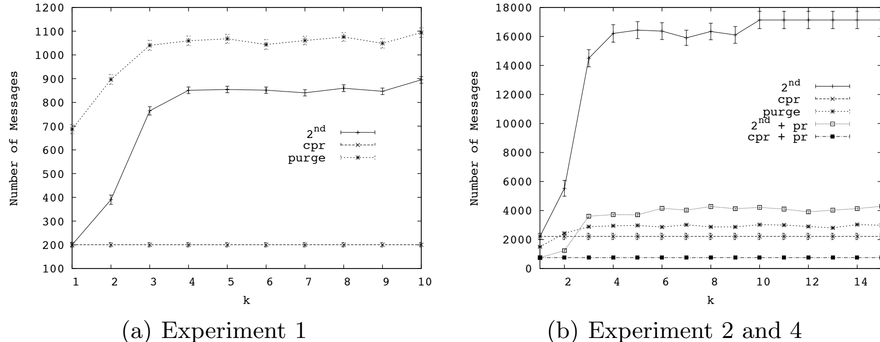
1. Before t', $\forall v \in V$ $\overrightarrow{min}_v$ and $dmatrix_v$ are correctly computed.
2. At time t', $\overline{v}$ is compromised and advertises $\overrightarrow{bad}$ (a vector with a cost of 1 to *every* node in the network) to its neighboring nodes.
3. $\overrightarrow{bad}$ spreads for a specified number of hops (this varies by experiment). Variable k refers to the number of hops $\overrightarrow{bad}$ spreads.
4. At time t, some $v \in V$ notifies all $i \in adj(\overline{v})$ that $\overline{v}$ was compromised. [6]

The message and time overhead are measured in step (4) above. The precomputation (Section 3.1) is not counted towards message and time overhead.

4.1 Fixed Link Weight Experiments

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

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



(a) Experiment 1 (b) Experiment 2 and 4

Fig. 2. Experiment 1, 2, and 4 plots. (a) Experiment 1 - message overhead for Erdös-Rényi Graphs with fixed unit link weights, where $n = 100$, $p = 0.05$, and diameter= 6.14. (b) Experiment 2 and 4 - message overhead for Erdös-Rényi graph with random link weights, $n = 100$, $p = .05$, and average diameter=6.14. The 2nd `best`, `purge`, and `cpr` curves correspond to Experiment 2. Experiment 4 additionally includes 2nd `best` + `pr` (2nd `best` using poison reverse) and `cpr` + `pr` (`cpr` using poison reverse).

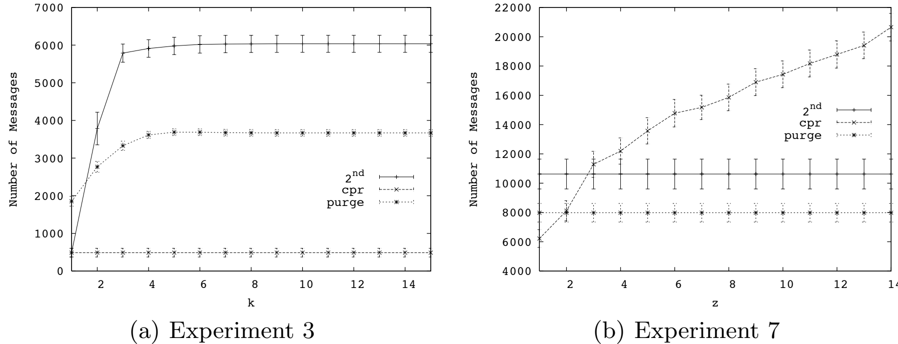
Experiment 1 We start with a simplified setting and consider Erdös-Rényi graphs with parameters n and p. n is the number of graph nodes and p is the probability that link (i, j) exists where $i, j \in V$. The link weight of each edge in the graph is set to 50. We iterate over different values of k. For 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 $n = 100$, $p = \{0.05, 0.15, 0.25, 0.25\}$, 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.

The results for this experiment are shown in Figure 2(a). We omit figures for $p = \{0.15, 0.25, 0.50\}$ because the results follow the same trends as $p = 0.05$ [9]. `cpr` outperforms `purge` and 2nd `best` because $\overrightarrow{bad}$ is removed using a single diffusing computation, while the other algorithms remove $\overrightarrow{bad}$ state through distance vector's iterative process.

With 2nd `best`, distance values increase 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. [7] Our profiling numbers indicate that there are few (if any) pairwise routing loops during 2nd `best` recovery, indicating that nodes quickly count up to their final least costs.

Although no pairwise routing loops exist during `purge` recovery, `purge` incurs overhead in its purge phase. Roughly, 50% of `purge`'s messages come from the `purge` phase. This accounts for `purge`'s high message overhead.

[7] We formally prove these properties in our technical report [9].



(a) Experiment 3 (b) Experiment 7

Fig. 3. Plots for Experiment 3 and 7. (a) Experiment 3 - Rocketfuel graph number 6461 with $n = 141$ and diameter=8. (b) Experiment 7 - message overhead for Erdös-Rényi with random link weights, $n = 100$, $p = 0.05$, $k = 2$, and $\lambda = 4$. z refers to the number of timesteps `cpr` must rollback.

`purge` and 2nd `best` message overhead increases with larger k. Larger k implies that false state has propagated further in the network, resulting in 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.

The trends for time overhead match those for message overhead. The interested reader can refer to our technical report [9] for these figures.

Experiment 2 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 2(b) shows the message overhead for $n = 100$ and $p = .05$. We omit the figures for the other p values because they follow the same trend as $p = .05$ [9].

In striking contrast to Experiment 1, `purge` outperforms 2nd `best` for all values of k. 2nd `best` performs poorly because the count-to-∞ problem: when $k < 4$, there are 1K pairwise routing loops (a strong indicator of the occurrence of the count-to-∞ problem) and over 10K routing loops occur for each $k \geq 4$. No routing loops are found with `purge` because they are removed by `purge`'s diffusing computations. `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.

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

Experiment 3 In this experiment, we simulate our algorithms over Internet-like topologies downloaded from the Rocketfuel website [3] and generated using GT-ITM [2]. We show the results for one Rocketfuel graph in Figure 3(a). The results follow the same pattern as in Experiment 2.

Experiment 4 We repeat Experiments 2 and 3 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. The results are shown for one representative topology in Figure 2(b), where 2nd `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, 2nd `best` + `pr` sees a significant decrease in message overhead when compared to the standard 2nd `best` algorithm because poison reverse removes the many pairwise routing loops that occur during 2nd `best` recovery. However, 2nd `best` + `pr` still performs worse than `cpr` + `pr` and `purge`. When compared to `cpr` + `pr`, the same reasons described in Experiment 2 account for 2nd `best` + `pr`'s poor performance. Comparing `purge` and 2nd `best` + `pr` yields interesting insights to the two different approaches for eliminating routing loops: `purge` prevents routing loops using diffusing computations and 2nd `best` + `pr` uses poison reverse. Because `purge` has lower message complexity than 2nd `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.

4.2 Link Weight Change Experiments

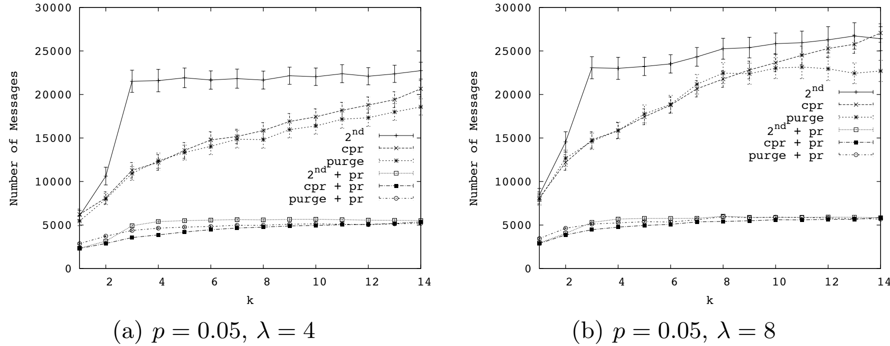
In the next three experiments we evaluate our algorithms over graphs with changing link costs. We introduce link cost changes between the time $\overline{v}$ is compromised and when $\overline{v}$ is discovered (e.g. during $[t', t]$). In particular, let there be λ link cost changes per timestep, where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

Experiment 5 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.

Due to space constraints, we only show results for $p = .05$ and $\lambda = \{4, 8\}$ in Figure 4. [8] `purge` yields the lowest message overhead, 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 $[t', t]$. In contrast, 2nd `best` and `purge` process some of the link cost change events during $[t', t]$ as part of normal distance vector execution. In our experimental setup, these messages are not counted because they do not occur in Step 4 of our simulation scenario described in Section 4.

We also find that 2nd `best` performance suffers because of the count-to-∞ problem. The gap between 2nd `best` and the other algorithms shrinks as λ increases because link cost changes have a larger effect on message overhead with increasing λ.

[8] Our experiments for different λ and p values, yield the same trends [9].



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

Fig. 4. Plots for Experiment 5 and 6. 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 $\lambda = \{4, 8\}$. Experiment 5 includes the 2nd `best`, `purge`, and `cpr` curves. The curves for 2nd `best` + `pr`, `purge` + `pr`, and `cpr` + `pr` correspond to Experiment 6.

Experiment 6 In this experiment, we apply poison reverse to each algorithm and repeat Experiment 5. 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 4, poison reverse improves `purge` performance. We include the three new curves in both Figure 4 plots (each new curve has label "`algorithm-name`" + `pr`). Results for different λ and p values yield the same trends and so we omit the corresponding plots.

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 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 $[t', t]$. As λ increases, the performance gap decreases because `cpr` + `pr` must process all link cost changes that occurred in $[t', t]$ while 2nd `best` + `pr` and `purge` + `pr` process some link cost change events during $[t', t]$ 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 checkpoint assumption.

Experiment 7 Here 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 $[t', t]$. Otherwise, our experimental setup is the same as Experiment 5.

Due to space constraints, we only display a single plot. Figure 3(b) shows the results for an Erdös-Rényi graph with link weights selected uniformly at random between $[1, n]$, $n = 100$, $p = .05$, $\lambda = 4$ and $k = 2$. We plot message overhead against the number of timesteps cpr must rollback, z. The trends are consistent when using the poison reverse optimization for each algorithm. 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.

4.3 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 be identified.

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.

5 Related Work

There is a rich body of research in securing routing protocols [10, 17, 20]. Unfortunately, preventative measures sometimes fail, requiring automated techniques for recovery. Previous approaches to recovery from router faults [15, 19] 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 et al. [12] 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 [12].

Database crash recovery [14] and message passing systems [6] 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 least costs are non-negative.

Garcia-Lunes-Aceves's DUAL algorithm [8] uses diffusing computations to coordinate least cost updates in order to prevent routing loops. In our case, `cpr` and the prepossessing procedure (Section 3.1) 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 [11] 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 `cpr`, 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.

6 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 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 finding the worst possible false state a compromised node can inject (e.g., state that maximizes the effect of the count-to-∞ problem). We have also started a theoretical analysis of our algorithms.

7 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. D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
5. E. Dijkstra and C. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, (11), 1980.
6. 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.
7. 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.
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. 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.
11. D. Jefferson. Virtual Time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.
12. P. Liu, P. Ammann, and S. Jajodia. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.
13. 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.
14. 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.
15. J. Moy. Hitless OSPF Restart. In *Work in progress, Internet Draft*, 2001.
16. V. Padmanabhan and D. Simon. Secure Traceroute to Detect Faulty or Malicious Routing. *SIGCOMM Comput. Commun. Rev.*, 33(1):77–82, 2003.
17. 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.
18. 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.
19. A. Shaikh, R. Dube, and A. Varma. Avoiding Instability During Graceful Shutdown of OSPF. Technical report, In Proc. IEEE INFOCOM, 2002.
20. 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.