INTRUSION-TOLERANT CLOUD MONITORING

by

Thomas J. Tantillo

A thesis submitted to The Johns Hopkins University in conformity with the requirements for the degree of Master of Science, Engineering.

Baltimore, Maryland

July, 2013

Abstract

Cloud computing is a new computing paradigm that offers immense cost benefits. As a result, a wide variety of services, including critical infrastructure, are migrating to the cloud. The problem of ensuring that cloud networking continues to operate correctly under all circumstances becomes important; downtime in cloud network communication equals downtime for critical services.

Accurate and timely monitoring is paramount for the correct operation of a cloud. Since cloud administrators are remote to their system, even if part of the cloud infrastructure is compromised, the monitoring system must remain operational in order for administrators to see and react to problems, prompting the need for an intrusion-tolerant messaging system.

We introduce Priority-Based Flooding with Source Fairness, an intrusion-tolerant message dissemination protocol that is designed to provide an intrusion-tolerant cloud monitoring solution. This protocol guarantees optimal resiliency and real-time delivery of messages in the presence of compromises, at the cost of flooding messages on an administrator-defined network topology.

We implement Priority-Based Flooding with Source Fairness in Spines, an open source overlay network framework. We evaluate our implementation on a simple network topology and a realistic network topology based on a global cloud provider's network. The evaluation shows that Priority Flooding maintains fairness among the active source nodes in the network, meets the desired bandwidth guarantee, and ensures real-time delivery of sources' higher priority messages even when the network resources are in full contention.

Acknowledgments

First and foremost, I thank my family for their unwavering support over the past twenty-five years of my life. My parents, Mary and Thomas "Chief" Tantillo, have been and continue to be my role models. My mother taught me to be a perfectionist in my work and encouraged me to always set high goals. My father has taught me, by example, to live each day with a positive attitude and that generosity goes a long way. I thank my twin brother and best friend, Nick Tantillo, for life-long camaraderie and for showing me how to be unselfish; Nick never hesitates to sacrifice his time and energy to help any of his close friends in need, and I constantly aim to live up to his standards.

I am immensely grateful to my advisor, Dr. Yair Amir, for all of his advice and support during my time at Hopkins. At a time when I was unsure of my career path, he persuaded me to do research and accepted me into the graduate program. Yair is an ideal mentor, displaying professionalism, enthusiasm, sensitivity, unselfishness, respect, and constant availability toward each of his students. Over the past three years of working with him, I have grown tremendously, both in the field of Computer Science and as a person. I can honestly say that I would not be where I am today without his help.

I am deeply indebted to my aunt, Claire Miko, and my uncle, Robert Dominicus, for all of their emotional and financial support. They have enabled me to pursue my dreams in a worry-free manner, and my seven years at Hopkins would not have been possible without them.

I thank Amy Babay, Ricky Sharma, Dr. Marco Platania, and Dr. Vladimir Braverman for the discussions and brainstorming sessions that we had with one another. They made every day in our lab both intellectual and enjoyable.

I thank my collaborators at Purdue University, Andrew Newell and Dr. Cristina Nita-Rotaru, for their invaluable advice regarding my research. They constantly provided a fresh and unbiased perspective on our work and ensured that it was always presented in the highest quality.

I thank John Schultz, Jonathan Stanton, Nilo Rivera, Raluca Musaloiu-E., John Lane, Jacob Green, and many of the other former members of the Distributed Systems and Networks lab that I interacted with over the past three years. They have always been more than willing to help me with my research and constantly make themselves available day and night, regardless of their own busy schedules.

ACKNOWLEDGMENTS

I thank Andrew Kesner, Lina Kim, and especially Tommy Harrelson, my roommates over the past few years. They spent many hours talking with me about my research and lent a helping hand whenever it was possible. Andrew, Lina, and Tommy have made the past three years fun and exciting each and every day.

I especially thank my girlfriend, Megan Hindle, for her unending love and support throughout the entirety of our four wonderful years together. Megan has become one of my closest friends and continues to keep me motivated in my graduate studies with her own amazing accomplishments. I cherish the time we spend together and eagerly look forward to our future.

Last but certainly not least, I thank Daniel Obenshain, without which this work could not be possible. Daniel has worked with me side by side during the entirety of my graduate studies at Hopkins. Without him, my success in the graduate program would be a mere fraction of what it is today. Daniel demonstrated how to be a professional world-class researcher and encouraged me to work hard every day. He is one of the most intelligent, motivated, and kind persons that I have ever met, and I consider him a true friend.

Thomas Tantillo
July 2013

This work was supported in part by DARPA grant N660001-1-2-4014. Its contents are solely the responsibility of the authors and do not represent the official view of DARPA or the Department of Defense.

Contents

Chapter 1

Introduction

Cloud computing offers a cost-effective and highly-available approach to run the world's IT infrastructure. Rather than building their own infrastructure from scratch, a wide variety of services are leveraging from pre-existing data centers and the widely-available Internet infrastructure to reap accessibility, economies of scale, and specialization benefits. The cloud computing trend can be seen in applications from all domains, ranging from an amateur's weekend project to critical infrastructure systems such as the supervisory control and data acquisition (SCADA) systems that control the power grid or water supply.

Although the cloud offers immense cost benefits, it also raises several security concerns. Applications that once ran on private, isolated networks now run on an infrastructure that is connected to the Internet, making it much easier for attackers to access both the application and the underlying infrastructure that the application depends on to function. Moreover, sophisticated compromises pose a real threat to clouds; the possibility for intrusions is not purely theoretical. "Stuxnet", "Duqu", and "Flame" are recent examples of sophisticated attacks that were specifically designed to target nuclear control systems [1, 2, 3]. The target systems were low-level, relatively simple compared with computers, and not connected to the Internet. In contrast, the network routers that comprise cloud infrastructure are more like computers and are connected to the Internet, making them more accessible targets. Additionally, variation among routers is limited by the low number of commercially-available solutions, which increases the motivation for an adversary, as a single successful attack may be able to compromise many routers.

With the attack surface of critical services growing, the problem of ensuring that cloud networking continues to operate correctly under all circumstances becomes important; downtime in cloud network communication equals downtime for critical services. Cloud messaging must persist through benign faults, physical problems, misconfigurations, software bugs, and even sophisticated intrusions, i.e., external breaches into the system with the ability to obtain valid credentials, insider attacks, and even collusion among several compromised system entities.

Accurate and timely monitoring is paramount for the correct operation of a cloud. Since cloud infrastructure nodes are geographically distributed in data centers and administrators are remote from their cloud components, administrators must rely on the underlying cloud infrastructure monitoring messages to provide status updates detailing the current state of the network. If the monitoring system fails for any reason, cloud administrators can find themselves blind to problems and unable to resolve issues. In essence, cloud administrators are faced with the classic chicken-

and-egg problem; the cloud monitoring system must work *at some level at all times*, even while under attack, in order for administrators to react and resolve problems.

Existing work in the area of resilient clouds includes network monitoring systems, intrusion detection systems, and intrusion-tolerant messaging. Network monitoring systems maintain the correct operation of clouds in benign environments by notifying administrators of failing or crashed components due to benign failures. However, the types of sophisticated compromises that we tolerate in this work are outside the threat model of state of the art network monitoring systems.

Intrusion detection systems or intrusion detection and prevention systems monitor networks for malicious activity, usually from attacks launched outside the network. State of the art intrusion detection systems are successful at identifying failed cloud components or broken network connections due to detectable malicious attacks. Similar to benign monitoring systems, the compromised components can be isolated and reported to cloud administrators for repair or replacement. In this work, we consider undetectable compromises, effectively making our threat model beyond the scope of intrusion detection systems and intrusion detection and prevention systems.

Existing intrusion-tolerant messaging systems effectively tolerate intrusions by detecting problems with end-to-end message delivery. Once the malicious behavior is detected, these systems take the time to isolate the location of the intrusion in the network and remove the tainted component. In this work, we aim to preserve the real-time aspect and availability of cloud monitoring and consider sophisticated compromises that are undetectable, and as a result cannot use state of the art intrusion-tolerant messaging systems.

1.1 Problem Description

We consider sophisticated compromises as external breaches into the cloud network with the ability to obtain valid credentials, insider attacks, and collusion among compromised system entities. Once a system component is compromised, we assume a Byzantine fault model where adversaries behave arbitrarily within the threat model, which we describe in detail in the threat model section (Section 2.2). We design our protocol to tolerate worst-case compromises, and thus consider crashes, misconfigurations, and other benign failures as strictly a subset of adversarial behavior.

Sophisticated intrusions can have negative effects on unprepared networks. A compromised router can drop, delay, or replay any messages that pass through it from neighboring routers. Moreover, since the compromised router has valid authentication credentials, it can generate malicious routing information and distribute it through the network, causing it to look more favorable as a forwarding choice for other nodes' shortest path calculations. As a result, the compromised router can launch the same message delivery attacks on more traffic than it normally would receive. In an extreme case, a black hole attack can be launched in which nearly all messaging is eliminated.

With knowledge of the harmful effects that compromises can have on global-scale cloud networks, an effective intrusion-tolerant messaging system for cloud monitoring must meet several criteria. The messaging system must defend the network from *all* of the potential actions a compromised node (or nodes in collusion) can perform. In addition, the system must meet the high-availability and real-time demands of cloud monitoring systems. The messaging system should provide optimal resiliency, delivering messages as long as a single path of correct nodes exists between source and destination. Note that if no such correct path exists, no messaging system can succeed. In order to give cloud administrators an up-to-date view of their cloud, the messaging system should pass the most important information in a timely manner. When network resources are constrained, a source's higher priority messages are preferred over that source's lower priority messages to maintain real-time delivery of the most critical information.

1.2 Solution Highlights

In this thesis, we present *Priority-Based Flooding with Source Fairness*, the first practical intrusion-tolerant monitoring system that provides optimal resiliency and timeliness, even in the presence of compromised nodes. Priority Flooding provides real-time and priority-based dissemination semantics, both of which are essential in monitoring systems. The protocol uses controlled authenticated overlay flooding, which leverages cryptographic authentication and overlay networks to flood messages across an administrator-defined network topology. In order to control the cost of Priority Flooding, an administrator can select a subset of the network topology, as opposed to the complete network topology, on which to flood messages. Thus, Priority Flooding provides a complete and practical intrusion-tolerant cloud monitoring solution.

Priority Flooding uses a concept we call *maximal topology*. In essence, the maximal topology uses authentication from public/private key pairs set up by an offline administrator to provide a maximum membership in the network and all the potential links between these members. Consequently, the maximal topology significantly limits the power of an adversary, as a compromised router can no longer generate false routing information that tries to create false links or nodes. Since we are using flooding to disseminate the monitoring messages, we can completely ignore advertised weights on the edges. Compromised routers can no longer advertise low edge costs in order to put themselves on other correct nodes' shortest path calculations because shortest path calculations are omitted from flooding schemes.

Priority Flooding provides optimal resiliency. As long as a single path of correct nodes exist between source and destination, messages will be delivered to that destination. Note that flooding is optimal because if no such path of correct nodes exists between a source and destination, no dissemination protocol can succeed in delivering messages. Moreover, Priority Flooding passes messages with optimal timeliness; messages travel on all correct paths, and thus the path with the lowest latency will deliver the message to the destination.

Although Priority Flooding is optimal in both resiliency and latency, it comes at a cost. To put flooding into perspective, the average cost to send a message via flooding compared with secure single-path routing is the number of edges in the network divided by the average path length. Initially, this overhead may seem massive. However, we claim that the cost is practical for two reasons. First, the Priority Flooding protocol does not flood messages on the bare Internet; rather, we use controlled authenticated overlay flooding. In this scheme, an overlay network topology is protected by an offline administrator's private key, and messages are flooded on the links of this topology. In our experience, a global cloud network infrastructure only requires tens of intelligently deployed overlay nodes to effectively cover the entire globe (see Section 2.1). A network administrator can further control the cost of flooding by explicitly selecting which of the overlay edges (and overlay nodes) from the complete topology will participate in the flooding. Second, the flooding overhead is not incurred for all types of messages; we only flood the important monitoring messages. Our experience with production cloud networks shows that monitoring messages make up a very small fraction of the overall network traffic. As a result, the tax for flooding monitoring messages remains small enough that cloud providers are willing to pay for flooding in order to give their monitoring system optimal availability and timeliness.

Since the Priority-Based Flooding with Source Fairness protocol was primarily motivated by the demands of cloud monitoring, it is not surprising that the protocol has a natural affinity for real-time and priority-based applications. The underlying goal of Priority Flooding is to maintain messaging between correct, direct neighbors no matter what compromised nodes do elsewhere in the network. To support this goal, we have two main assumptions: (1) compromised nodes cannot prevent correct, direct neighbor nodes from communicating, and (2) the cryptographic primitives that we use for authentication will not be broken.

The Priority Flooding protocol guarantees "Source Fairness." We want to ensure that compromised nodes cannot starve resources from correct source nodes. However, it is impossible to determine whether a source node injecting a lot of messages is maliciously trying to consume bandwidth or correctly reporting a lot of necessary information. As a result, the only way Priority Flooding can ensure fairness is to treat each active competing source node equally. The naive approach is to split the total available bandwidth into preallocated equal fragments. However, Priority Flooding instead takes a more intelligent approach; bandwidth is allocated on a packet-by-packet granularity. Each active competing source node gets either all the bandwidth it requests or its fair share (the total amount of bandwidth divided by the number of active competing source nodes).

With the underlying liveness and fairness properties in place, we now explain the semantics that the Priority Flooding protocol provides. In typical network monitoring systems, each node periodically reports its status by sending messages to a destination, where these messages are collected to try and give an administrator a "global" view of the system. Thus, in our protocol, every node can potentially be a source node. The status messages from a source are not all equal; some convey more important information than others. In normal situations, when network resources such as bandwidth are not constrained, Priority Flooding can deliver all of a source's messages in a timely manner. However, when network resources become limited, e.g., due to contention, it is crucial that we maintain timely delivery by only sending the most important messages. Since Priority Flooding cannot determine if any of the sources are malicious, it cannot naively choose to send the highest priority messages it has stored across all sources. In this case, a malicious source could send all of its messages with highest priority to try to obtain preferential treatment over other correct sources. Instead, Priority Flooding considers message priority independently for each source. The protocol continues to deliver as many of a source's higher priority messages that it can without sacrificing the real-time nature, and the remaining lower priority messages from that source must be dropped.

An interesting byproduct that we obtain from the Priority Flooding protocol is a new protocol, *Priority-Based Multicast with Source Fairness*. In the regular Priority Flooding protocol, messages are already delivered to all correct nodes. Leveraging from this fact, instead of sending messages destined to a single node, source nodes in Priority Multicast send to a multicast group. Priority Multicast provides the same availability and timeliness guarantees as Priority Flooding. One difference between Priority Multicast and Priority Flooding is the network requirements necessary for successful delivery. Recall that in Priority Flooding, we need a correct path of nodes between the source and destination. In Priority Multicast, successful delivery is determined independently for each member of a target multicast group. For each member, if there is a correct path of nodes between the source and that member, messages can be delivered to that member.

In order to ensure that Priority-Based Flooding with Source Fairness (and Priority Multicast) operates correctly in a real-world setting with potentially malicious components, we implement the Reliable Intrusion-Tolerant Link protocol to manage every connection in the network. The Reliable Intrusion-Tolerant Link is required to protect correct nodes from malicious neighbors that may try to consume a correct node's network resources (see Chapter 4).

We implement the Priority-Based Flooding with Source Fairness protocol and the Reliable Intrusion-Tolerant Link protocol in the open source Spines overlay messaging framework, publicly available at www.spines.org. Spines enables the creation of custom dissemination protocols at the overlay level for easy and rapid deployment. The modular nature of the Spines framework allows us to incorporate both our Priority Flooding dissemination and Reliable Intrusion-Tolerant Data link.

With the Spines-based implementation of Priority Flooding, we evaluate the protocol on emulated networks. Using several different network scenarios, including a topology based on a global cloud provider's network, we show that Priority Flooding meets the high-availability and real-time demands of cloud monitoring systems, while ensuring fairness among all source nodes at all times.

The key contributions of this work are as follows:

- Introducing Priority-Based Flooding with Source Fairness, the first practical intrusion-tolerant monitoring system that provides optimal resiliency and timeliness, even in the presence of compromised nodes.
- Implementing the Priority-Based Flooding with Source Fairness protocol in the open source Spines overlay messaging framework (www.spines.org).
- Evaluating the Priority-Based Flooding with Source Fairness protocol in emulated networks, including a realistic topology based on a global cloud provider's network (www.ltnglobal.com).

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- The next section presents related work in the areas of network monitoring systems, intrusion detection systems, securing routing protocols, and intrusion-tolerant messaging systems.
- Chapter 2 defines the model of the cloud, the threat model, our assumptions, model definitions, and desired guarantees of the intrusion-tolerant cloud monitoring system.
- Chapter 3 presents Priority-Based Flooding with Source Fairness, our intrusion-tolerant messaging engine that provides real-time cloud monitoring in the presence of intrusions. A description of the protocol, data structures, system parameters, and psuedocode for the protocol is provided. Additionally, we discuss several trade-offs and prove the desired guarantees of our protocol.
- Chapter 4 presents the Reliable Intrusion-Tolerant Link protocol that protects Priority Flooding nodes from potentially malicious neighbors.
- Chapter 5 describes implementation considerations for the Priority Flooding protocol, including the advantages of message expiration, the Spines overlay messaging framework, and the necessary changes to make Spines intrusion-tolerant in order for Priority Flooding and the intrusion-tolerant link protocol to be incorporated into the architecture.

- Chapter 6 presents an evaluation of Priority-Based Flooding with Source Fairness on several emulated network topologies, including a realistic topology based on a global cloud provider's network.
- Chapter 7 concludes the thesis.

1.4 Related Work

The work in this thesis touches primarily on four main areas of focus: network monitoring systems, intrusion detection systems, securing routing protocols, and intrusion-tolerant messaging. We next discuss the related work that has been done in each area.

1.4.1 Network Monitoring Systems

Network monitoring systems are successful at maintaining the correct operation of clouds in benign environments, where cloud components may become slow or crash due to benign failures. Network monitoring systems constantly monitor the network, detect failing components, and notify cloud administrators so that these components can be fixed or replaced.

Nagios [4] is often considered as the industry standard in IT Infrastructure Monitoring. Nagios is an open source network and computer monitoring system that alerts administrators both when problems arise and when problems are solved. Nagios offers a extensive list of services for network monitoring with the goal of minimizing system downtime. For example, Nagios monitors network and host services, monitors host resources, offers remote monitoring via SSH tunnels, and notifies administrators when system components have failed or are about to fail.

Cacti [5], *Zabbix* [6], and *Ganglia* [7] are other popular network monitoring systems. Cacti is a web-based network monitoring and graphing tool designed to be the front-end for RRDtool, an industry-standard data logging tool. Cacti users can poll network services at defined intervals and graph the resulting data, such CPU load and network bandwidth utilization. Zabbix is enterprise-class network monitoring tool that is designed to monitor the status of network services and components. Zabbix can be installed on network hosts or can be sent status updates (e.g., via TCP, SSH, etc.) to monitor statistics such as CPU load, disk space, and network utilization. Ganglia is a scalable distributed monitoring system for clusters, Grids, and other high-performance computing systems. Ganglia is based on a hierarchical design in order to minimize the overhead incurred at each monitored machine. It allows an administrator to remotely view live or recorded statistics such as CPU load or network utilization.

Nagios, Cacti, Zabbix, and Gangila have seen tremendous success in benign environments, but these monitoring systems cannot monitor networks in the face of the types of sophisticated intrusions that we consider in this thesis. These network monitoring systems rely on information that is collected from trusted network components. With this implicit trust, even a single intrusion can cause the entire network infrastructure to fail. A compromised network component can launch attacks on the network's routing protocol (see Section 2.5) in order to completely disrupt routing and render the network monitoring system useless.

1.4.2 Intrusion Detection Systems

There is extensive work in the area of Intrusion detection systems and intrusion detection and prevention systems. Intrusion detection systems aim to protect a network against malicious activity by continually monitoring a network for malicious symptoms, which usually stem from attacks launched from outside the network. Intrusion detection and prevention systems go one step further, responding to a detected threat by trying to prevent it from succeeding. State of the art intrusion detection systems are immensely successful at identifying compromised cloud components, which can then be isolated and reported to cloud administrators for repair or replacement.

Snort [8] is a popular intrusion detection system that is used in both production and research systems. Snort is a packet sniffer and logger that is based on libpcap and acts as a lightweight network intrusion detection system. It contains rules that match content patterns to detect attacks such as buffer overflows, port scans, worms, and much more. Snort separates itself from other packet sniffers because it can perform in-depth packet payload inspection. Snort can be given rules to detect malicious activity that is specific to the application layer.

Common packet analyzers such as the popular *tcpdump* [9] run on the command line and allow the interception of transmitted and received traffic by hosts on a network. tcpdump is commonly used to analyze network behavior and performance. For the purpose of intrusion detection, tcpdump can monitor a network's infrastructure to determine whether expected routing is occurring properly, and if not, allows an administrator to isolate a problem.

Another popular form of intrusion detection systems are host-based intrusion detection systems, which run on individual network hosts. *OSSEC* [10] is an open source host-based intrusion detection system that performs log analysis, file integrity checking, rootkit detection, and real-time alerting for the underlying host machine.

OSSEC can provide intrusion detection for most operating systems and is widely used by Internet service providers, universities, and governments.

Although popular intrusion detections systems like Snort, tcpdump, and OSSEC have seen incredible success, they cannot provide the highly-available and timely cloud monitoring service that we require in this thesis. In our threat model, adversaries can compromise cloud network components and launch a variety of sophisticated attacks on unprepared networks. Compromised nodes may selectively forward or drop incoming messages, severely deteriorating the quality of service of messaging. Such an attack may be undetectable by intrusion detections systems because it appears as a simple, yet persistent benign network problem. A more sophisticated and harmful attack is when a compromised node distributes malicious routing information or colludes with other compromised nodes in the network to launch a black hole attack, where a large portion of the network-wide traffic is sucked into this node and dropped, effectively halting the flow of the affected traffic (see Section 2.5). Detecting the cause of this attack and restoring correct network operation is difficult, as the compromised node(s) have valid security credentials and performed legal actions with respect to the network's routing algorithms. Even if an intrusion detection system could isolate the problem, the downtime in detecting and fixing the problem raises major practical concerns for the network's monitoring messages. If these messages are part of the traffic that is affected by the black hole attack, monitoring simply does not exist until the attack is resolved, clearly violating the availability and timeliness guarantees we require.

In contrast, our work in this thesis provides built-in defenses against these types of network routing attacks by eliminating the negative effects the attacks can have on the overall messaging system. Our protocol successfully delivers messages, even in the presence of the most sophisticated compromises, as long as there exists a path of correct nodes between a source and destination. Note that no message dissemination algorithm can do better. Therefore, our work continuously provides the availability and timely requirements of cloud monitoring under all possible circumstances.

1.4.3 Securing Routing Protocols

The attacks that compromised routers can launch against naive network routing protocols has been studied extensively for decades. Previous work proposes a variety of solutions to secure routing against malicious activity.

In Finn et al. [11], the authors use public/private key pairs and shared secret keys to provide authenticated routing updates and reduce the vulnerability of dynamic networks to external attacks. The shared secret keys establish a secure connection on each link between two routers and allows updates to be encrypted and authenticated across links, preventing an outsider from injecting malicious updates or tampering with messages traveling on that link. Public key authentication is done with the help of a trusted third party such as a Certificate Authority, and enables routers to verify that a router who claims it sent an update was actually the router that introduced that update. Finn et al. also investigates the validity and integrity of the routing updates passed through the network. They identify that compromised routers have the credentials to legitimately create routing updates advertising false links or false nodes that can severely damage the network if forwarded and applied. In order to prevent this damage, these malicious updates must be detected and removed. Finn et al. mentions that having the complete topology of the network stored at each router will detect many of the malicious updates, but admits that this approach is infeasible due to the scalability concerns of large networks. Instead, they offer two different approaches. First, a regional authority can manage all routers in a certain region and is responsible for validating routing updates. All messages are first sent to the corresponding regional authority for validation before they can be sent toward the target destination. Second, intermediate routers can validate pieces of an update they have sufficient knowledge for as the update is forwarded through the network. By the time the update reaches its destination, the entire update is either validated or invalidated.

Despite the advancements Finn et. al made in securing routing protocols, the work cannot adequately provide cloud monitoring in the presence of sophisticated compromises. While the work tries to identify malicious routing updates that advertise nonexistent nodes or edges, having every message in the network sent first to a regional authority for validation imposes significant latency overhead and creates a throughput bottleneck in the network. Cloud monitoring is a demanding service that requires low latency and prefers high throughput. In addition, if one of the regional authorities is compromised, routing updates can no longer be authenticated. For their second approach, validating pieces of a routing update as it passes through intermediate routers fails if multiple compromised routers collude to validate a malicious update with false information. Furthermore, Finn et. al does not mention how to defend the network against malicious routing information that launches black hole attacks. A single compromise can advertise low-weight edges to suck in and drop a large portion of the network traffic, potentially disrupting network routing.

In contrast, in this work we defend cloud networks against malicious routing information by using a maximal topology, which explicitly enumerates all the cloud nodes and links between them. Unlike the networks discussed in Finn et al., our cloud model (see Section 2.1) uses overlay networks with tens of nodes, allowing knowledge of the complete, maximal topology to be feasibly stored at each cloud node. Therefore, the cloud network is immune to black hole and other routing attacks. Our Priority Flooding protocol achieves optimal latency while ensuring that all messages are authenticated.

In Kumar and Crowcroft et al. [12], the authors recognize that network routing protocols operate in a vulnerable environment. If routing protocols are not properly protected from malicious attacks, adversaries can subvert them by modifying, deleting, or adding false information in routing updates. Their work integrates security into inter-domain routing protocols similar to the Border Gateway Protocol [13]. Kumar and Crowcroft et al. presents an inter-domain routing protocol that defends against attacks on links between routers. For example, the inter-domain routing protocol uses encrypted checksums to prevent tampering of messages, uses sequence numbers to prevent replayed routing updates, and uses timeouts between border gateways to prevent delay attacks.

Kumar and Crowcroft et al. cannot defend a network against the types of sophisticated router compromises that we discuss in this thesis. Similar to BGP, their inter-domain routing protocol relies on an inherent trust model between routers. If one or more of the routers is compromised, it can completely disrupt networking by launching a routing attack like the black hole attack. In contrast, we explicitly design our messaging protocol to withstand router compromises and continue passing messages at all possible times.

In Murphy et al. [14], the authors defend the Open Shortest Path First (OSPF) [15] link-state routing protocol using digital signatures. Routing information is signed with an asymmetric private key at the source and is verified at each router recipient to check the integrity and authenticity of the update.

The work in Murphy et al. cannot defend the network against sophisticated router compromises. In the control plane, a compromised router has valid credentials to sign and distribute malicious routing information that can cause a black hole attack. In the data plane, compromised routers can act as source nodes and spam messages into the network, causing resource starvation from other correct source nodes. In contrast, our intrusion-tolerant messaging protocol provides a complete solution. We use digital signatures to ensure the integrity and authenticity of messages injected into the network by each source. In addition, the network by design is immune against control plane attacks such as the black hole attack. Lastly, we provide defenses against compromised source nodes that spam messages into the system by ensuring that no active source node can starve resources from another active source node in the network (see Section 2.4).

In Bradley et al. [16], the authors present WATCHERS, a protocol based on the principle of conservation of flow in a network to detect and react to subverted routers that drop or misroute packets. Correct routers running WATCHERS keep count of the number of packets entering and leaving neighboring routers as well as which links packets leave routers. If a good router detects that one of its neighbor's counters is incorrect or that a neighbor routes packets incorrectly, the router cuts the communication with that neighbor and deems it as incorrect.

Although Bradley et al. presents an intrusion detection approach for securing network routing, the work contains practical and performance barriers that prevent it from providing a cloud monitoring solution that ensures optimal resiliency and adequate timeliness. The practical barriers stem from assumptions that place strict limitations on where network compromises can occur. WATCHERS assumes that any bad router has at least one good neighbor and that all correct routers are connected by a path of correct routers. These assumptions guarantee that correct routers on either side of a bad router can exchange packet counters. In addition, WATCHERS assumes that the network contains a majority of correct routers at all times. If any of the above assumptions are not met, WATCHERS cannot guarantee that maliciously behaving routers will be detected and removed from the network. For the performance barriers, when a counter mismatch is detected, WATCHERS must spend time to locate and remove links to bad routers, during which normal messaging, and thus cloud monitoring, cannot continue.

In contrast, we design our messaging protocol to tolerate any number of compromised nodes; we do not assume that a majority of correct nodes exists nor do we require that incorrect nodes have correct neighbor(s). In an extreme scenario, our protocol will successfully deliver messages from a source to destination as long as they are connected by a path of correct nodes, and even if the nodes in this correct path are *the only* correct nodes in the entire network. Furthermore, we ensure optimal timeliness for the delivery of messages because messages are guaranteed to traverse the minimum-latency correct path between any source and destination.

In Cheung et al. [17], the authors present protocols that detect and respond to misbehaving routers causing denial of service attacks. Cheung et al. presents two approaches for challenging neighbors, *Distributed Probing* and *Flow Analysis*. In the distributed probing protocol, routers periodically challenge their neighbors with special test packets and compare their neighbors' behavior against the expected behavior in order to detect if a router is acting maliciously. The test packets are constructed to look like data packets in order to prevent a bad router from distinguishing test packets from regular packets. The flow analysis protocol is a predecessor of the work in Bradley et al. [16] mentioned above. For each router, that router's neighbors collaborate their incoming and outgoing packet counters to diagnose the router.

Similar to Bradley et al., Cheung et al. cannot provide an effective cloud monitoring solution due to practical barriers. The work assumes that the expected behavior of a router's neighbors is always known and that each set of neighbors has a consistent view of the network, including connectivity and shortest path calculations. These two assumptions cannot be guaranteed in practice because the types of networks Cheung et al. considers relies on disseminated routing information to update each router's view of the network, which may lead to temporary inconsistencies. In addition, Cheung et al. has the same assumptions as Bradley et al. that limit the location and distribution of compromises in the network, with the addition that no bad routers are neighbors. Lastly, the work assumes there are no benign link losses or failures since they can give the impression that a malicious router is dropping messages. On wide area networks, including typical cloud network infrastructures, benign packet loss is not uncommon.

In contrast, our messaging protocol can tolerate any number of compromised nodes and all types of failures, including benign link failures that cause packet loss. Messages are delivered from source to destination as long as a path of correct nodes exists between that source and destination. We do not make any assumptions about the behavior of a node's neighbors. Since our protocol uses flooding, nodes do not rely on disseminated routing information to update routing decisions. All correct nodes in the network have a consistent view of the network topology due to the maximal topology, which enumerates all of the cloud nodes and the links between them.

1.4.4 Intrusion-Tolerant Messaging

There is a variety of work in the intrusion-tolerant messaging systems area, where the goal is to continue to pass messages even when part of the network is compromised. Next, we discuss previous messaging systems that are most similar to our work in this thesis.

In Perlman et al. [18], Perlman presents a scheme for protecting link-state updates of a routing protocol by using public keys for authentication. Trusted nodes supply network routing nodes with the ID and public key associated with each other routing node in the network. Messages are signed with a source's private key and are verified with that source's public key when they are received at other nodes. Perlman uses flooding to propagate the link-state updates through the network, which is the dissemination approach that link-state routing protocols commonly use. In order to ensure fair link utilization, each router in the network maintains a separate buffer for the latest link-state update from each source and uses a round-robin transmission scheme between the buffers. Perlman mentions that rather than storing a single message for each source, each router can allocate up to M messages. All link-state updates are sequenced at the source node and the buffers use overtaken-by-event semantics, where the latest link-state update is always stored for each source. If a received update is newer than the one stored in the buffer, it overwrites the older one in memory. If an older update is received, it is simply discarded.

Perlman et al. cannot provide a practical cloud monitoring solution. In the work, each router preallocates a fixed-sized buffer for each source in the network. As a result, a source node sending in the network receives a constant $\frac{1}{n}$ fraction of the total resources at each router in the network (where n is the total number of source nodes in the network), regardless of the number of current competing source nodes. Ideally, the total resources at each router would be assigned only to active source nodes. Moreover, each intermediate router in Perlman's work only keeps the newest message(s) from each source. It is not uncommon for a source node to inject a burst of messages into the network during a peak in activity or in response to interesting events. If the burst of messages does not fit entirely in the fixed buffers at intermediate routers, the oldest packets are dropped. However, some of these older, dropped messages may be more critical in the information they convey than the newer, stored messages. Ideally, the protocol would allow the newest, most important messages to be stored at intermediate nodes. Otherwise, there is no guarantee that the most critical information is delivered to the destination.

In contrast, we design a practical messaging protocol that meets the dynamic properties of cloud monitoring. Unlike in Perlman et al., only active source nodes contend for resources at each intermediate correct node; we do not preallocate a fixed number of messages per source, allowing for full memory utilization at all times. In addition, our protocol does not naively keep the latest message received from a source, since that message may not be as important as a previously received message. In order for intermediate nodes to determine which messages are most important and should remain in memory, our protocol uses a source-defined priority level on each message. If memory is in contention and a newer message arrives, it overwrites the oldest, least important message from the same source node, and only if such a message exists. Otherwise, the newer messages is simply dropped. These properties allow our protocol to continuously and reliably deliver the highest priority messages allowed by network resources in a timely fashion.

In Awerbuch et al. [19, 20], the authors present an on-demand routing protocol for ad hoc wireless networks that is resilient against byzantine failures. On-demand routing protocols differ from normal routing protocols in that route discovery is only performed when data packets need to be routed. The protocol presented in Awerbuch et al. is a source-based routing scheme that operates in three-phase cycles. In the first phase, a source node uses the link-state information it receives from flooded routing updates along with a faulty link weight list (see phase three below) in order to calculate the least-weight path to its target destination. In the second phase, messages are routed normally along the chosen path and the source monitors the path for byzantine faults by using an adaptive probing technique. The probing identifies a faulty link between two potentially byzantine nodes after $\log n$ faults have occurred, where n is the number of hops in the path. In the model, faults occur when acknowledgements of data packets are not received within a predetermined time. Since faults can be due to either benign packet loss or byzantine behavior, the chosen time should be proportional to the expected benign packet loss rate on the path in order to reduce the false positive rate. The probing detection scheme is kept hidden from byzantine nodes because the probing requests are placed on actual data packets. Byzantine nodes cannot selectively drop data packets without also dropping the probing information. Finally, in the third phase, the discovered faulty link is added to faulty link weight list and the its weight is multiplicatively increased. Since the link's weight is only reduced back to normal slowly over time, the source greatly reduces the chance that this link will be chosen as part of paths in the near future. At this point, the protocol continues from the first phase, finding a new path between the source and destination with the updated weights.

The on-demand routing protocol is optimal in terms of throughput when no problems are detected, as messages are only sent along a single path. However, when problems are present and detected, the protocol has practical limitations that prevent it from being a good candidate for a real-time cloud monitoring system. First, the protocol suffers latency during the path probing that is required to isolate a faulty link. An adversary that simultaneously controls k network nodes can disrupt network communication by a factor of $b \cdot kN \cdot log^2(n)$, where b is the message loss threshold for deeming a link as faulty, N is the total number of nodes in the network, and n is the upper-bound on the non-faulty path length in the network. Since the protocol is designed for ad hoc networks, each of the k compromised nodes can advertise a link to all other $N-1$ nodes in the network. Each iteration of the three-phase cycle only increases the weight of a single faulty link, and therefore the process must be repeated $O(kN)$ times to remove all faulty links and restore network communication. To make matters worse, the $b \cdot kN \cdot log^2(n)$ factor can be repeated by the adversary once the weights of the kN faulty links are reduced to their normal values. Moreover, since the protocol relies on source-based routing and sources do not share their faulty link weight list, this latency is suffered independently at each correct source in the network. Second, Awerbuch et al. assumes that network connectivity is stable along the chosen path in the second phase of each cycle. Without connectivity, the protocol cannot route messages along the path nor accurately detect and probe byzantine behavior.

In contrast, our protocol constantly incurs bandwidth overhead because monitoring messages are flooded, rather than sent along a single lowest-weight path. However, our protocol maintains network availability and real-time delivery of monitoring messages from each correct source node at all times. As long as a path of correct nodes exists between a source and destination, an adversary cannot disrupt network communication nor cause latency overhead. Moreover, our protocol does not assume stable connectivity and can deliver messages even if connectivity is sparse by using eventual path propagation.

Amir et al. [21, 22] presents *Authenticated Adversarial Routing*, a messaging protocol that will successfully route packets from a source to destination as long as a correct path of nodes connects them in each round. The work assumes that such a path exists, but does not assume which nodes are compromised nor which path is correct in each round. Amir et al. uses public-key authentication to provide integrity for a source's messages and achieves optimal asymptotic throughput, i.e., the authors prove that no protocol can be asymptotically superior in terms of throughout.

Authenticated Adversarial Routing is based on the Slide protocol [23], where packets travel from high pressure nodes to lower pressure nodes through the network, analogous to the flow of water. The source node maintains high-pressure by continually introducing new packets and the destination node acts as a sink (with zero pressure) by continually clearing packets as they arrive. Initially, only the source node has pressure built up; all the other nodes have zero pressure. Since packets only flow across an edge that has lower pressure on the other side, the network must reach steady state, with the appropriate pressure built up at each intermediate network node, before packets will reach the destination. The first packets sent from the source fill the buffers at intermediate nodes, create the steady state pressure, and remain stuck unless there is a change in the network topology that alters the steady state pressure of a node. The total capacity of the pressure data structures is $4n^3$. Each of the n nodes in the network can hold at most $2n$ packets from the source along a maximum of $2n$ directed edges at any time. Once steady state is reached, every packet introduced by the source results in the delivery of a single packet to the destination.

When a problem is discovered, Authenticated Adversarial Routing achieves fault localization using *Routing with Responsibility*. Every node must sign key parts of packets they transfer to their neighbors in order to be held accountable for their actions. When a problem is detected at the destination, e.g., the number of received packets is below the expected amount, the sender is notified to request status reports from all nodes in the network. If a node refuses to answer, it is put on the blacklist and deemed corrupt. Otherwise, with a complete status report, the sender can look through the key parts of the packets that were signed by each node in order to identify and eliminate a corrupt node.

Despite its optimal throughput and resiliency properties, Authenticated Adversarial Routing cannot provide a practical intrusion-tolerant cloud monitoring solution. In each round, the protocol incurs $O(n^3)$ overhead to build up pressure in the network and $O(n^4)$ overhead to identify and eliminate a corrupt node from the collected status reports. Moreover, only a single message can be sent in each round. If a source has knowledge of a large message a priori, e.g., in the case of a file transfer, the overhead to build up pressure is reasonable compared with the overall message size. In this case, optimal asymptotic throughput is achieved for most of the message. However, in cloud monitoring, the messages are relatively small and periodic. Both the latency and pressure build-up overhead is completely impractical for each set of periodic monitoring updates injected by a cloud source node. Finally, Authenticated Adversarial Routing assumes that the network is synchronous, which poses limitations for real-world deployment.

In contrast, our work in this thesis is specifically designed for cloud monitoring requirements, continuously maintaining network availability and real-time delivery of monitoring messages. Priority Flooding does not require initial messaging overhead nor latency before messages can be delivered. Rather, messages are delivered as soon as a source node injects them in the network. Our protocol does not achieve optimal asymptotic throughput. The cost of our protocol is "the number of edges in the network divided by the average path length" times more than secure single-path routing. However, given the typical size of well-distributed global cloud networks (see Section 2.1) and the fact that cloud monitoring messages make up only a small percentage of the overall traffic, the cost of our protocol is acceptable for the benefits it provides. The combination of acceptable cost and natural affinity for cloud monitoring makes our solution both effective and practical.

In Deng et al. [24], the authors introduce *Intrusion-Tolerant Routing for Wireless Sensor Networks* (INSENS). INSENS is similar to our work we present in this thesis because it tolerates intrusions by bypassing malicious nodes rather than trying to detect intrusions. In order to protect the network's routing information, INSENS assumes that the base station is trusted and is the only entity allowed to set up or alter routing information. Since INSENS operates on wireless sensor networks and compromised routers can only affect nearby neighbors, INSENS is able to confine an intrusion to a small vicinity in the network. INSENS sets up multiple paths through disjoint parts of the network so that even if an intruder takes down one path, secondary paths exist to forward packets to the correct destination.

Although the intrusion tolerance aspect of INSENS relates to our work in this thesis, the wireless sensor network nature does not support cloud monitoring. In cloud network infrastructure, the effect of intrusions cannot be confined to a limited vicinity in the network. There is no trusted base station that is the only network entity able to change routing information and flood messages. Cloud networks are fully-distributed systems. Each network node has the capability to broadcast messages and disseminate routing updates. Since the nodes have wired connections, a single compromised node can affect the entire cloud network. For these reasons, we design our protocol in this thesis to support cloud monitoring while defending against the threat model associated with cloud network infrastructure.

Alternative Dissemination Approaches to Flooding

The protocol presented in this thesis uses controlled authenticated overlay flooding to disseminate monitoring messages thorough the network. By using flooding, we obtain both optimal resiliency and latency. It is noteworthy to compare flooding with two other potential dissemination approaches: secure authenticated single-path routing and K node-disjoint paths routing. In secure authenticated single-path routing, a single lowest-cost path connecting the source and destination is selected. Messages are then only sent on this path. Single-path routing is less costly than flooding since messages are only sent along a single path. Flooding is actually "the number of edges in the network divided by the average path length" times more expensive than single-path routing. If all of the nodes in the single path are correct, messages will be delivered with the corresponding latency of the path. However, if even a single node on the path is compromised, correct messaging is no longer guaranteed.

In K node-disjoint paths routing, the source calculates K node-disjoint paths from the source to destination and sends each message a total of K times, once along each node-disjoint path. In terms of resiliency, K-paths routing correctly delivers messages to the destination as long as at least one of the node-disjoint paths consists entirely of correct nodes. If an adversary is able to compromise even a single node on one of the node-disjoint paths, that path fails. In the worst case, a clever adversary can compromise K nodes, with one compromise on each node-disjoint path, to completely cut communication between the source and destination. Therefore, K-paths routing is provably secure against up to $K - 1$ compromises. Compared to flooding, K-paths routing is less resilient. The $K - 1$ compromises are determined by the minimum cut of the network topology between the specified source and destination. In contrast, flooding guarantees optimal resiliency and can tolerate compromises as long as a path of correct nodes exists between the source and destination. In terms of latency, K-paths routing will deliver the message to the destination along the lowest-latency correct path from the set of calculated K node-disjoint paths. In flooding, messages are delivered along the correct, global-minimum latency path in the entire network. It is certainly possible that the global-minimum latency path is among the paths chosen by K-paths routing. In this case, if no nodes on this path are compromised, both K-paths routing and flooding guarantee the same optimal latency. However, if one of the nodes on this path is compromised, the latency guarantee of each protocol differs. Flooding will automatically find the new global-minimum latency, correct path. In contrast, K-paths will deliver messages along the correct next-best latency path among the set of remaining chosen node-disjoint paths, which is not necessarily guaranteed to be the new global-minimum latency, correct path in the network.

Chapter 2

Model

In this section, we define the network model of the cloud, the threat model we consider, and our assumptions. In addition, we present the desired guarantees that our cloud monitoring protocol, Priority-Based Flooding with Source Fairness, will provide at all times, even in the presence of cloud compromises. Finally, we present several known network attacks and how our messaging protocol and model successfully defend cloud monitoring against them.

2.1 Cloud Model

Clouds usually span several geographically-distributed data centers in order to constantly provide cloud clients with low-latency access and to ensure resilience against geographically-correlated faults. The cloud networking infrastructure consists of cloud nodes, logical communication links between these nodes, and cloud clients (see Figure 2.1 below). Each cloud node is deployed at one of the data centers and communicates directly with its immediate cloud neighbors. Two cloud nodes are neighbors if they are located within data centers that are directly connected in the cloud infrastructure. A pair of cloud nodes that are not direct neighbors must communicate with each other via a path of other cloud nodes, where each hop in the path is between two directly connected nodes. All cloud links are bidirectional.

Cloud clients are processes that use the cloud networking infrastructure for messaging. The clients can be internal cloud processes that help manage and control the cloud or external processes that interact with the cloud's service model. Cloud monitoring is an example of internal cloud clients. The source clients are internal processes that generate critical status updates. Updates are collected in a timely fashion at the destination client, another internal process that logs and performs machine learning on these updates. In order to establish a more resilient connection, cloud clients connect to several nearby cloud nodes. Clients continuously monitor these connections, and if the connection quality of the active cloud node falls below the required threshold, clients seamlessly migrate, sending their messages to another cloud node that provides satisfactory service.

In our model, all of the cloud nodes have a consistent view of the complete cloud network topology. An offline cloud administrator defines the maximal topology (i.e., all of the cloud nodes and potential links) a priori, signs the topology with their private key, and distributes the topology to all of the cloud nodes. Each cloud node knows the public key of the offline administrator as well as the public keys for all cloud nodes

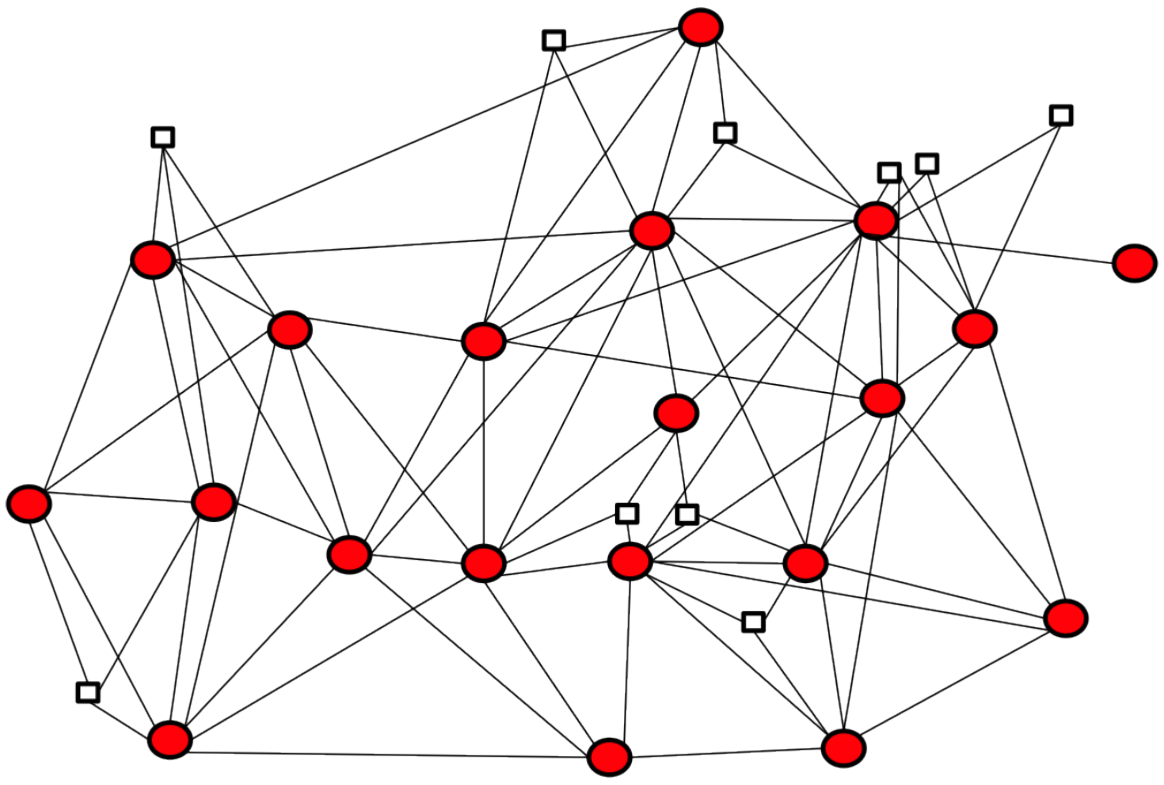


Figure 2.1: Network Model of the Cloud. Each cloud node is deployed at one of the geographically-distributed data centers (red circles). Cloud clients (white squares) connect to several of the nearest cloud nodes in order to establish a resilient connection.

in the network. This allows each cloud to verify the integrity and authenticity of both the maximal topology and data messages throughout the lifetime of the cloud system. In addition to providing all cloud nodes with a consistent view of the cloud topology, the maximal topology also defends the control plane of the network by significantly limiting the power of a compromised cloud node. Compromised nodes can no longer successfully distribute malicious routing information to create a false node or link in the network. All correct nodes in the network adhere to the connectivity defined in the maximal topology, only accepting messages from and forwarding messages to legitimate neighbors.

In order to update the cloud topology with new nodes or edges, the offline cloud administrator can update the maximal topology and resign it. In this sense, the offline administrator is converted into a Certificate Authority. The updated maximal topology and any new necessary information (e.g., the public key of a new cloud node) can be redistributed to all cloud nodes in the network. All correct cloud nodes will only communicate with other nodes that have adopted the newest maximal topology, which can be verified by comparing a hash value of the latest topology's representation along with the offline administrator's signature on the latest topology.

Since our cloud network model relies on a maximal topology, we do not consider ad-hoc or wireless networks. In such networks, membership and connectivity is dynamic. New members may announce themselves or may be discovered by existing network nodes, and these new members can join the network without prior authentication by an offline administrator. In intrusion-prone settings, these new members may be adversarially controlled, and if an unlimited number of them are allowed to join the network, they may overwhelm the bandwidth or computational resources of correct network nodes. Similarly, we do not consider multi-administrative domain networks (e.g., the Internet) connected via inter-domain routing protocols (e.g., BGP). In such networks, no single administrator has the authority to manage and constrain the entire network topology. In addition, the routing decisions are based on network policies and rule-sets. In intrusion-prone settings, this inherent trust model is unacceptable, since compromised nodes in one domain can have devastating effects on the routing of other correct domains.

2.2 Threat Model

In this work, we anticipate sophisticated compromises and consider a Byzantine threat model, where an attacker that compromises a cloud node can behave arbitrarily. Attackers have access to a compromised node's private cryptographic keys stored in memory and can generate signed messages from the node that will properly verify at other cloud nodes. Compromised nodes may execute modified versions of both our messaging protocol's code and other code running on the node. An attacker can eavesdrop on the traffic between correct nodes and can launch message attacks at a compromised cloud node that drop, delay, reorder, replay, duplicate, and insert packets in the network.

We assume that attackers have full knowledge of our messaging protocol and its intricate details, including static parameters, such as protocol timeouts for message retransmission or fault detections, and dynamic parameters that change throughout the lifetime of the protocol. In addition, attackers can distinguish between different packet types. As a result, they can try to forge messages or tamper with specific fields in the headers of individual packets that are routed through a compromised node. Additionally, a compromised node can selectively forward or drop messages and forward messages to the wrong destination node. Lastly, an attacker can have a compromised node behave correctly from the point of the view of other nodes in the network.

We consider collusion attacks, where a set of attacker-controlled nodes establish an out-of-band connection in order to work together to try to subvert our messaging protocol. We assume that attackers have large network bandwidth and ample amounts of memory and computation, potentially much more than correct network nodes.

We do not assume a bound on the number of nodes an attacker can compromise simultaneously. Our messaging protocol guarantees liveness, i.e., delivery of messages from a source cloud node to destination cloud node, as long as a correct path of cloud nodes exists between that source and destination. However, the number of compromised nodes can affect the throughput and latency guarantees of our protocol (see section 2.4 below), as these guarantees depend on the number of active source nodes in the network during network contention

In our model, a *correct* cloud network node is one that executes our messaging protocol faithfully. In contrast, an *incorrect* cloud network node is any node that is not correct. An *incorrect* cloud node may be compromised by an adversary and can exhibit arbitrary behavior described above.

2.2.1 Assumptions

In this work, we have two main assumptions.

1. Two correct nodes that are direct neighbors in the cloud network topology can exchange messages, even in the presence of incorrect nodes. Incorrect nodes cannot launch denial-of-service attacks that completely consume computational or network resources between two correct cloud nodes.

2. Attackers are computationally bounded such that they cannot break the cryptographic constructs used by our messaging protocol. We require that adversaries cannot break encryption, forge digital signatures, or find collisions in secure hash functions.

2.3 Model Definitions

Next, we specify definitions of our model that we will use in stating the desired guarantees of the Priority Flooding protocol (section 2.4).

Node. *A node is a single logical participant in the protocol. A node has a single private key and a single ID to identify its messages at all other nodes in the network.*

Each node is viewed as a single entity by all the other nodes in the network and is allocated resources accordingly. Each node has a single private key which it uses to sign all its messages. The associated public key is known by all other nodes in the network, for the purpose of verifying these signatures.

Link. *A link is a set of two nodes.*

Links are bidirectional and either node on the link can send messages to the node on the other side.

Maximal Topology. *A maximal topology is a set of nodes, a set of links between those nodes, and the public key associated with each node.*

The maximal topology is the maximal set of nodes that can participate in the protocol and the maximal set of allowed links between those nodes. At any given time, the nodes currently participating in the protocol may be a strict subset of the maximal set of nodes, due to benign failures or malicious compromises. Similarly, the current usable links may be a strict subset of the maximal set of allowed links, due to network disconnectivity. However, a correct node will not agree to accept messages from a node to which it does not have a link in the maximal topology. Likewise, a correct node will not accept messages originating at a node not in the maximal topology.

Link Bandwidth. *The bandwidth of a link is the rate at which messages may be sent from one node in that link to the other.*

The rate at which messages can be sent on a link is determined by the underlying physical network. Occasionally, the link bandwidth may drop to zero if the link experiences disconnectivity. Due to the overhead of the headers and cryptographic signatures used in Priority Flooding, the bandwidth that a link receives is actually less than the bandwidth of the underlying physical network. See Section 6.1 for further information.

Link Latency. *The latency of a link is the sum of the propagation delay of the link and the worst-case queuing delay. The worst-case queuing delay is* $\frac{N * \textit{size of a message}}{\textit{link bandwidth}}$*, where N is the number of nodes defined in the maximal topology.*

The time between when a node receives a source's message from a neighbor and the node starts to forward one of that source's messages to a different neighbor is no greater than the worst-case queueing delay. Since sources are served in a round-robin fashion, the worst-case queueing delay occurs when a source waits for a node to forward a message from all other $N - 1$ sources first. The time to forward each message is $\frac{\text{size of a message}}{\text{link bandwidth}}$, and thus the worst-case queueing delay is $\frac{N * \text{size of a message}}{\text{link bandwidth}}$. After the worst-case queueing delay, a source's forwarded message arrives at the subsequent neighbor once the propagation delay of the link in the underlying network has elapsed, assuming that the message was not lost due to a network omission. Thus, the time between when a source's message arrives at a node and when one of that source's messages arrives at a neighbor node is no greater than than the sum of the worst-case queueing delay and the neighboring link's propagation delay.

Note that in the above description, there is no guarantee that the message that arrives at the node is the same message that gets forwarded on to another neighbor node within the bounded time. However, we can make such a guarantee in a specific case. As long as a source does not have any highest-priority messages stored at a node, if that source sends a highest-priority message to that node, the time between when that node receives the highest-priority message and when the same highest-priority message arrives at the next neighbor is no worse than the sum of the queueing delay and propagation delay. In this case, we can make the guarantee because intermediate nodes forward a source's messages in order of highest to lowest priority.

Path. *A path is an ordered set of nodes $a_1, a_2, a_3, \ldots$, such that all nodes in the set are in the maximal topology and each pair of nodes $\{a_i, a_{i+1}\}$ is a valid link in the maximal topology.*

Since each pair of adjacent nodes in the path is a valid link in the maximal topology, it is possible for messages to flow along this set of nodes, with each node forwarding the message to the next node in line. Note that the number of links in a path is one less than the number of nodes in the path.

Correct Path. *A correct path is a path where all nodes in that path are correct nodes.*

If some of the nodes on a path are incorrect nodes, there is no guarantee that the path will forward messages, since the incorrect nodes can simply drop those messages. In contrast, a path of all correct nodes is guaranteed to follow the protocol faithfully.

Path Bandwidth. *The bandwidth of a path is the minimum bandwidth over all links in that path.*

The maximum rate at which messages can be sent along a path is limited by the bottleneck link on that path. For this reason, the bandwidth of a path is exactly the bandwidth of the link with minimal bandwidth over all links in that path.

Path Latency. *The latency of a path is the sum of the link latencies in that path.*

The time it takes to send a message along a path is determined by the time it takes to transmit the message across each subsequent link. Therefore, the path latency is exactly the sum of the link latencies of each link in that path.

Shortest Correct Path. *A shortest correct path between a source S and destination D is the correct path from S to D with the minimum path latency.*

If there are many correct paths from S to D, one of them must have the minimum path latency and we choose the shortest path based on that metric. Using the minimum path latency as the metric supports the desired real-time properties of cloud monitoring systems.

2.4 Desired Guarantees

We now define the desired guarantees of the Priority-Based Flooding with Source Fairness protocol.

Theorem 1. *If there exists a correct path from a correct source node S to a correct destination node D, then the rate at which S can send messages to D is no less than $\frac{1}{n}$ times the bandwidth of that path.*

As long as S and D are connected through correct nodes via currently connected links, messages will flow from S to D. On the other hand, if incorrect nodes cut the network such that there is no correct path, then there is no guarantee that messages will flow from S to D. Lastly, if there are multiple correct paths from S to D, this guarantee is given independently for each, meaning that the rate at which S can send messages to D is at least $\frac{1}{n}$ times the maximum bandwidth over all the correct paths from S to D.

Theorem 2. *If the network has no highest-priority message from a source S, then if source S introduces a single highest-priority message into the network, the destination D will receive that message within some time t, where t is no greater than the path latency of the shortest correct path.*

This guarantee builds on the observation that we made when defining link latency. Recall that if a single node N does not have a highest-priority message from a source S stored, then if S sends a highest-priority message m to N, the same highest-priority message m will leave N and arrive at the next node within the link latency, or the sum of the link propagation delay and worst-case queueing delay. Looking at the bigger picture, if none of the correct nodes in the network have a highest-priority message m stored for source S, then the time between when each of the correct nodes receives m and when m arrives at a subsequent neighbor is bounded by that neighbor's link latency. Since we can bound the latency for the highest-priority message to traverse each correct link, and the shortest correct path is composed exclusively of correct links, we can guarantee that the latency of the highest-priority message is no worse than the sum of link latencies in the shortest correct path, or simply the path latency of the shortest correct path.

2.5 Protocol Behavior Against Attacks

Our Priority Flooding protocol will meet the resiliency and timeliness guarantees as long as the assumptions are met, regardless of the types of attacks launched by adversaries. In this section, we illustrate the behavior of our protocol in the face of common attacks to illustrate the protocol's effectiveness.

Tampered Routing Information

Thus far, we have emphasized intrusions and the significant impact they can have on networks that are not designed with intrusion tolerance in mind. However, even if an adversary is unable to compromise a router, they may be able to attack a routing protocol by targeting the routing information messages that are exchanged between nodes. Spoofing, replaying, or altering routing information can disturb network traffic, cause errors, partition the network, etc.

Protocol Behavior - In our model, we defend the cloud network against tampered routing information in two ways. First and foremost, all messages sent from cloud nodes are signed with that node's private key. Since each cloud node has the public key of all other cloud nodes, a node's messages are only accepted at other cloud nodes if the digital signature on each message passes validation. This public key authentication prevents an attacker from successfully tampering with messages traveling on a wire between two cloud nodes. Second, our messaging protocol is flooding-based, and therefore does not rely on routing information passed throughout the network to successfully deliver messages.

Selective Forwarding / Black Hole

Networks with source-destination pairs that are not directly connected operate with the assumption that participating routing nodes on the path between them will faithfully forward messages. If one of the nodes on the path is controlled by an adversary, the malicious node may refuse to forward certain messages. Naturally, one of the simplest forms of this attack is when the node acts like a black hole, dropping all messages it receives.

Protocol Behavior - It is impossible to force a compromised node to forward messages correctly. Instead, our messaging protocol overcomes compromised nodes by flooding messages throughout the network. As long as even a single path of correct cloud nodes exists between a source and destination, messages will be successfully delivered at the destination.

Sinkhole Attack

The sinkhole attack is essentially a combination of cleverly crafted routing information and selective forwarding or black hole attacks. In the sinkhole attack [25], an adversary tries to lure a majority of network traffic through a node it has compromised. Typically, the compromised node can be made to look attractive to surrounding nodes with respect to their routing algorithms by falsely advertising favorable routing information. With the sinkhole in place, a selective forwarding or black hole attack can easily be launched. While black hole attacks may be discovered by network administrators relatively quickly, selective forwarding is not as straightforward and may be harder to detect.

Protocol Behavior - Since our flooding protocol does not use advertised link weights in routing information to deliver messages, our protocol is immune to sinkhole attacks by design.

Wormhole Attack

In the wormhole attack [26], two colluding adversaries can understate their distance from each other by relaying messages through an out-of-band channel. As a result, a victim node that would normally see the two compromised nodes as far apart instead chooses to route packets along the path that includes them. Once again, at this point selective forwarding or a black hole attack can be launched. The wormhole attack has more sophistication compared with the other attacks discussed up to this point because the colluding routing node will also vouch for the false routing information.

Protocol Behavior - Similar to our protocol's behavior in the presence of sinkhole attacks, our flooding protocol does not use advertised link weights in routing information to deliver messages. Thus, our protocol is immune to wormhole attacks.

Chapter 3

Priority-Based Flooding with Source Fairness

The priority-based flooding with source fairness protocol is primarily motivated by the rigorous real-time demands of monitoring systems used in cloud infrastructure. These monitoring systems are responsible for distributing continuous streams of messages across a network, usually to some a few centralized locations where analysis can take place. Monitoring messages convey current status information from the perspective of a cloud node, and thus any node in the network can potentially be a source. In many applications, some monitoring messages are more important than others, as they contribute more critical information to produce an up-to-date picture of the overall network status. When resources are constrained, it is vital that the monitoring system maintains the real-time aspect of the updates. However, it may not be possible to deliver all of the messages given these limited resources. In such a scenario, the monitoring system should pass as many of the higher priority messages as possible, while potentially sacrificing, i.e., never delivering, some of the less critical, lower-priority information.

In this chapter, we describe the design and implementation of our intrusion-tolerant messaging system that retains the real-time and priority-based dissemination semantics of typical cloud monitoring systems, and continues to pass messages even in the presence of sophisticated compromises. First, we provide a high-level description of the Priority-Based Flooding with Source Fairness protocol. Then, we present a detailed description, including the data structures used in the protocol, the message types, and pseudocode. Finally, we discuss trade-offs and use the description of the Priority Flooding protocol to prove that the desired guarantees from Section 2.4 will be met under the network model.

3.1 High-Level Description

Priority-Based Flooding with Source Fairness is a flooding-based dissemination algorithm that combines specialized data structures and cryptographic authentication to guarantee optimal resiliency and real-time delivery of messages. To provide optimal resiliency, messages are flooded through the network, ensuring that each packet travels on all possible paths between the source and destination. As long as one of these paths is correct, the packet will be delivered. To support real-time delivery, Priority Flooding is designed to allow every node in the network to make a fast forwarding decision with each incoming message. When a node receives message, it must quickly determine if this message is new and should be forwarded to neighbors, or if the packet is a duplicate and can simply be discarded.

In general, each Priority Flooding data structure falls into one of two categories: those that are common across all of a node's links and those that are maintained independently on each outgoing link. The primary data structure shared across all links is a hash table that stores uniquely received messages and their metadata, including the source node, sequence number, and priority. Whenever a node receives a message, it can quickly perform a lookup in the hash table to determine whether or not that message is unique. Note that this hash table provides a necessary defense against message replay attacks, where a malicious node forwards old messages from another source in order to consume that other source's buffer space. The hash table enables nodes to recognize duplicate messages and simply discard them rather than incorrectly storing them in the victim source node's memory buffers.

The second group of data structures, those that are maintained independently on each of a node's outgoing links, consists of two main data structures: a fair sending queue and a message priority queue. The fair sending queue ensures fairness between active source nodes for which a node has messages to forward. All active source nodes are placed in the queue and selected in a round-robin fashion. When a source node is selected, choosing which stored message from that source to send is determined by the message priority queue. Since each message in the network is assigned a source-defined priority level before being injected into the network, the messages can be ordered by importance on each outgoing link. The message priority queue maintains a pointer to the highest and lowest priority message from each active source, and when prompted to send a message it returns this highest priority message. If a link's message priority queue exceeds the maximum capacity, the link finds the source that has the most messages stored and drops that source's oldest, lowest priority message.

Because our protocol is designed to operate in an intrusion-prone environment, cryptographic authentication is vital for the success of the protocol. Priority Flooding uses cryptographic authentication to prevent spoofing and identify message tampering. The authentication is network wide and is based on standard public-key cryptography. Each node in the network is assigned a public/private key pair and knows the public key of all other nodes (via the maximal topology that is signed and distributed by an offline network administrator). Correct source nodes digitally sign each message with their respective private key before injecting the message into the network. Intermediate forwarding nodes and the destination node can verify the signature to ensure that the message was not altered and originated from the advertised source.

3.1.1 Fairness

Fairness is an important component of any dissemination scheme, as it guarantees that no source is starved of network resources. In intrusion-prone environments, fairness is particularly vital because adversarial sources may try to obtain preferential treatment over other correct sources. In Priority Flooding, fairness is maintained separately on each link in the network since every link independently manages which messages are forwarded to the corresponding neighboring node. Sources are served on each link in a round-robin fashion using the fair sending queue, and only one message is forwarded each time a source is selected.

In contrast to static bandwidth allocation schemes that reserve a predetermined amount of bandwidth for each potential source in the network, Priority Flooding dynamically allocates bandwidth based on the number of currently active sources, or the number of sources from which a link has messages stored to send. By not preallocating bandwidth, Priority Flooding enables full utilization of each link's bandwidth capacity at all times. Thus, Priority Flooding is fair if each competing entity, i.e., each currently active source node, receives either everything it requests or its fair share of the resources on each link between two correct nodes.

To illustrate the Priority Flooding fairness property, consider two examples on a link with 100 Mbps bandwidth capacity and three active source nodes A, B, and C. Each source's fair share of the link bandwidth is $33\frac{1}{3}$ Mbps. In the first example, A, B, and C all request a rate of 50 Mbps. Since each of their requests is higher than their fair share, the sources will only be allocated $33\frac{1}{3}$ Mbps each. In the second example, A requests 10 Mbps, B requests 50 Mbps, and C requests 60 Mbps. Since A now requests less than its fair share, the link allocates 10 Mbps for A. The remaining $23\frac{1}{3}$ Mbps that A is not using is split fairly between B and C, resulting in both sources being allocated 45 Mbps. In both examples, all sources are allocated the minimum of their request and their fair share of the link's bandwidth, and the protocol achieves full link utilization.

It is important to understand why Priority Flooding allocates resources fairly on a packet-by-packet basis. In the presence of sophisticated compromises, there is no way for a node to determine which sources, if any, are compromised. A source that is sending a lot of traffic may be compromised or may just be a correct node with a burst of information to deliver. The only resource allocation strategy that eliminates the possibility of malicious interference is one that allocates bandwidth fairly. Moreover, allocating on a packet-by-packet granularity is crucial because it prevents malicious sources from temporarily starving other sources, which is something that can happen in credit-based allocation schemes. In these schemes, a malicious source can keep quiet for an extended length of time, build up a large amount of credit, and then use the credit all at once to prevent other sources' messages from flowing.

3.1.2 Timeliness

Priority Flooding is designed to deliver messages to their destinations in a timely manner. Ideally, the latency of messages are only affected by the propagation delay of the underlying physical network, and the time spent at each intermediate node in the network is minimal. Since Priority Flooding's data structures are designed to support quick forwarding operations, the only cause of latency at a node is the time a message spends waiting in the message priority queue before being sent on the link. When the rate that messages enter a node is less than or equal to the rate that messages leave a node, messages are delivered with propagation-delay latency, since none of the messages wait in a link's queue for long.

In reality, the opposite case is often true, where the rate of messages entering a node is strictly greater than the rate of messages leaving. There are many factors that can cause this type of behavior, including malicious source nodes spamming messages, correct source nodes sending a burst of messages, or fluctuating link bandwidth constraints. If messages are received faster than they can be forwarded, link message priority queues will begin to build up and eventually may become full. With messages now spending time in queues, the end-to-end latency of the messages will rise. Unfortunately, it is impossible to ensure that all of the queued messages are delivered with low latency. Priority Flooding draws from the monitoring system inspiration and uses two techniques to deliver as many important messages as possible with low latency.

First, even a link's message priority queue is full, it continues to receive messages. If a received message is newer or higher priority than one of that source's stored messages, it will be inserted in the queue in place of that source's oldest, lowest priority message. Of course, this means that not all messages a node receives are delivered. But, this is an appropriate trade-off to make for timeliness. In general, timeliness and reliability are often on opposite ends of the delivery-semantics spectrum. This technique constantly gives each node the most important up-to-date messages to forward.

Second, Priority Flooding sacrifices delivering less important messages in a timely manner in favor of guaranteeing that the more important messages get delivered in real-time. Priority Flooding sends higher priority messages first in order to ensure that they experience the lowest possible latency. Lower priority messages are either delivered with higher latency or dropped because newer higher priority messages arrived before these lower ones could be forwarded. This behavior is consistent with the goal of monitoring systems. It is preferable to deliver critical up-to-date messages in a timely manner than to deliver all the messages in the order in which they are received.

3.1.3 Priority Multicast

An interesting byproduct that we obtain from the Priority Flooding protocol is a new protocol, *Priority-Based Multicast with Source Fairness*. In the regular Priority Flooding protocol, messages are already delivered to all correct nodes. Leveraging from this fact, instead of sending messages destined to a single node, source nodes in Priority Multicast send to a multicast group. Priority Multicast provides the same availability and timeliness guarantees as Priority Flooding. One difference between Priority Multicast and Priority Flooding is the network requirements necessary for successful delivery. Recall that in Priority Flooding, we need a correct path of nodes between the source and destination. In Priority Multicast, successful delivery is determined independently for each member of a target multicast group. For each member, if there is a correct path of nodes between the source and that member, messages can be delivered to that member.

3.2 Detailed Description

3.2.1 Data Structures

The data structures used in the Priority Flooding protocol are designed to enable nodes to quickly and efficiently process incoming messages, determine which message to send next to a neighbor, and decide which message to drop when an outgoing link's memory is in contention. Each data structure falls into one of two categories: those that are common across all of a node's links and those that are maintained independently on each outgoing link.

Node-Common Data Structures

Incarnation

Upon startup, each node saves the time in seconds since the Unix epoch as its current incarnation. Additionally, each node maintains the most recent incarnation it receives from every other source in the network. The purpose of the incarnation is to provide a measure of freshness. Priority Flooding is designed to operate correctly in situations where nodes crash and restart, and the protocol state does not require stable storage in non-volatile memory. After a restart, a node's data structures will be reinitialized to default values. If the crash is not detectable by other nodes in the network, the recovered node may be misinterpreted as behaving maliciously and ignored. Consider the effect of the crash-recovery on a source node's messages. After a restart, the source node may unknowingly assign previously used sequence numbers to new messages. Other correct nodes in the network will see these messages as duplicates and discard them. The incarnation enables other nodes to recognize when a node has crashed and restarted. The combination of an incarnation and sequence number uniquely identify sources' messages in the network.

Sequence Number

In a given incarnation, every correct source node must ensure that each message it injects into the network is assigned a unique sequence number. Each node continuously maintains the sequence number to assign to its next message. For simplicity, the sequence number starts at 1 and is monotonically increasing. If a source reaches the maximum sequence number allowed, rather than wrapping around back to 0, the source performs an incarnation change and starts the sequence number from 1 again. This design choice prevents duplicate (incarnation, sequence number) pair packets from being injected by the same source.

Storage for Sources' Messages

The purpose of the message storage data structures is to provide efficient storage of received messages and to offer fast data structure operations, i.e., inserts, look-ups, and deletes. When a node receives a message on a link, it should be able to quickly determine whether the message has been received before. If the message is a duplicate, the node can simply set a flag and discard it. If the message is unique, the node should quickly insert the message into the source's corresponding data structure. Finally, when a node needs to forward a message to one of its neighbors, it should be able to efficiently obtain a reference to the message.

In order to meet the efficient storage and data structure operations requirements, every node maintains a separate hash table for each source in the network to store received messages and the respective metadata for those messages. Since the hash tables are common to all of a node's links, the message payload and metadata can be stored just once at each node. In addition, data structure operations on hash tables, including inserts, look-ups, and deletes, are all executed in $O(1)$ (constant) time.

Entries in the hash tables are key-value based. For a given source node, the key of each entry in that source's hash table is the (incarnation, sequence number) pair that uniquely identifies that source's messages. The value of the entry is the message payload and all of the relevant metadata for that message. The metadata consists of the message length, origin time at the source node, source-defined priority level, and the current status of this message toward each of the node's neighbors. For each neighbor, the status of the message indicates one of the following: the message is still needed by this neighbor, the message was received from this neighbor, the message was sent to this neighbor, or memory contention forced this message to be dropped on this link.

When a node receives a message from a source for the first time, it inserts an entry into that source's hash table and updates the status of this message toward all of its neighbors. At this point, the message is usually marked as needed by all neighbors except the one which the message was received from, which instead is marked as received. Over time, the message status toward each neighbor changes and is updated accordingly. If a copy of the message is received by a neighbor that needed the message, the status is changed to received. As soon as the message is sent toward a neighbor, the status is changed to sent. Similar behavior occurs when messages are dropped.

When a message is no longer needed by any of a node's neighbors, the message payload can safely be deleted because the node will never need to forward the message in the future. However, it is not safe to discard the metadata due to the possibility of replay attacks. If a correct node no longer needs to store a message and the metadata is destroyed, the same message can be replayed by a malicious neighbor. The correct node will think the message is unique, breaking fairness guarantees and wasting computational, memory, and bandwidth resources on the replayed message.

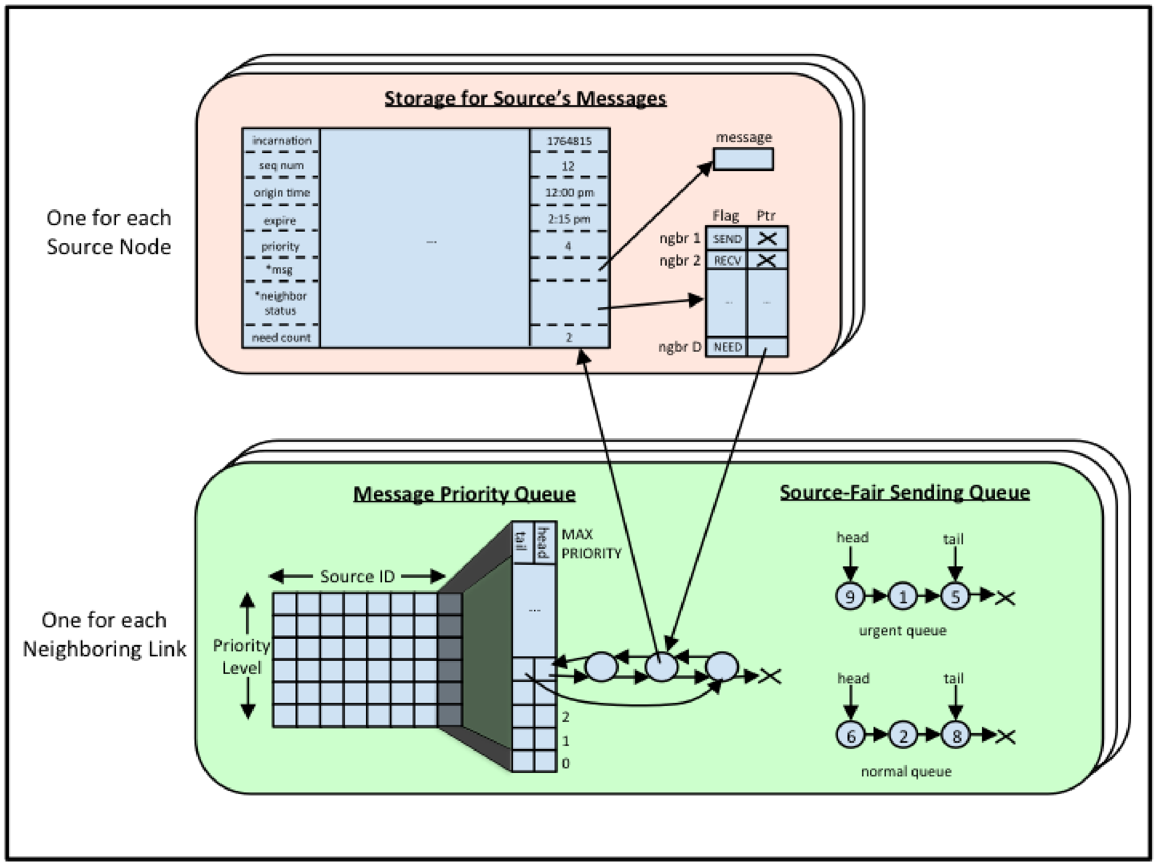


Figure 3.1: Priority Flooding Data Structures

Link-Specific Data Structures

Fair Sending Queue

In order to ensure that each source is given a fair allocation of a link's resources, each link maintains the fair sending queue data structure. In essence, the fair sending queue serves sources in a round-robin fashion, sending only one message each time a source is selected. Once served, the source is moved to the back of the queue to guarantee that every other source has an opportunity to send a message before the original source is served again.

A naive solution for the fair sending queue uses a single queue that contains all potential source nodes in the network. Whenever a link needs to send a message, it starts at the beginning of the queue and searches for the first source node for which this link has a message to forward. Once a source node is found, it is moved to the back of the queue. While this solution is simple, a link incurs large overhead when the number of active sources is small compared to the total number of potential sources in the network. In such a scenario, the link must traverse through the majority of the $O(n)$ queue each time it wants to send a message to find the next source in line.

In contrast to the naive solution, we design the fair sending queue used in Priority Flooding to provide links with fast and efficient querying. To reduce overhead, only active sources for which a link has messages to forward are present in the fair sending queue. In addition, when a previously inactive source becomes active, we want to give preference to that source's first message sent on the link compared with the messages of other already active sources. To support this behavior, the fair sending queue is composed of two distinct queues, an urgent queue and a normal queue. When a message is received and the source is not in either the urgent or normal queue, the source is inserted at the end of the urgent queue. When sending a message, the link serves the source at the front of the urgent queue. If the urgent queue is empty, it instead serves the sources at the front of the normal queue. In either case, the served source is moved to the back of the normal queue. If a source at the front of the normal queue does not have any messages to send, it is removed from the fair sending queues and will be added to the urgent queue as soon as a message from it arrives. Our fair sending queue ensures that infrequently sending sources are treated preferentially, but also guarantees that a malicious source cannot gain more than its fair share of a link's resources.

Message Priority Queue

The message priority queue is a link-specific data structure that plays an important role in several aspects of the Priority Flooding protocol. For forwarding messages to neighbors, the message priority queue actually supplements the fair sending queue. While the fair sending queue is responsible for serving active source nodes fairly, the message priority queue is responsible for keeping track of which message to send next for the chosen source. Similarly, when memory is in contention on a link, the message priority queue keeps track of which message to drop from the source that is currently consuming the most memory.

Similar to the fair sending queue, each link maintains its own message priority queue. Each message priority queue contains an array of FIFO (first in, first out) doubly-linked queues for every source in the network. The size the array is determined by the number of priority levels defined in the protocol (see Section 3.2.3). Thus, each link's message priority queue actually contains N * number_of_priority_level queues, where N is the maximum number of potential source nodes in the network. When a node receives a message, it determines which of its neighbors the message needs to be forwarded to and creates an entry in each link's message priority queue. This corresponds to inserting a priority queue node in the source's array of queues, indexed at the priority of the message. Each priority queue node points to the entry before and after it, and contains a pointer to the message payload that is stored in the node-common hash tables. This allows constant-time access to the message without incurring the overhead of storing the message more than once. In addition to managing a priority queue node for each stored message, each link keeps track of the total number of messages in the queue, as well as the number of message stored for each source.

By maintaining individual FIFO queues for each source and priority level, a link can easily keep track of the maximum and minimum priority level queue that it currently has a entry in for each source. When the link needs to forward a message for a source, it simply accesses the head of that source's maximum active priority level queue. Similarly, when the total number of messages exceeds the link capacity (a system parameter, see Section 3.2.3), the link can quickly scan all of the sources, find the source that has the highest number of stored messages, and then in constant time drop the message at the head of that source's minimum active priority level queue. Designing the message to support extremely quick look-ups is essential for the real-time nature of Priority Flooding.

It is important to understand why the message priority queue maintains a separate array of queues for each source node, rather than have only a single queue for each priority level shared across sources. Since the priority level of messages are source-defined, a malicious source may opt to send all of its messages with highest priority in order to gain preference over a correct source's messages with lower priority. With only a single queue for each priority level, choosing which messages to send would unfairly favor the malicious source, and choosing which messages to drop would unfairly disfavor the correct source. Thus, it is vital that we only compare priority levels of messages from the same source node.

3.2.2 Message Types

Since Priority Flooding is simple by design, the protocol only contains a single type of message, the data message.

Data Messages

A data message contains the actual data payload to be flooded as well as additional information in a protocol-specific header. The header includes the source of the message, the source's incarnation at the time of message creation, the message sequence number, and the priority set by the originating node. The message payload and header contents are protected by a digital signature that is generated at the source.

3.2.3 System Parameters

Priority Flooding contains several tunable system parameters that affect the performance or memory footprint of the protocol (see Section 3.3). A network administrator can tune the parameters as needed to achieve the desired behavior. For consistency, every correct node in the network uses the same value for each system parameter.

Link Capacity

The message priority queue that each link maintains can hold a maximum of *link_capacity* messages across all the sources in the network. As an example, if the link_capacity is set to 100 and two sources are saturating the link with messages, each source will on average be allocated space for 50 of their highest priority messages. We assume that the maximum size of the message is also bounded, and thus the memory footprint of each link is then link_capacity * the maximum message size. For a node, the memory footprint is node_degree * link_capacity * maximum message size.

Priority Levels

Each link's message priority queue stores a separate array of queues for each source node in the network. The number of entries in the array is determined by the number of priority levels that are defined by the administrator. The source nodes that inject messages into the network can assign each message a priority level between 1 and *num_priority_levels*. If an administrator sets up many priority levels, sources can assign very fine-grained priority differences for messages, but the array of queues grows in size. On the other hand, if there are only a few priority levels defined, sources can only assign coarse-grained priority differences, but the array of queues remains small in size.

3.2.4 Pseudocode

Event_Handler() - processes protocol events

```
1: Initialize()
2: while true
3:     switch event
4:         case received new message $m$ from connected source client
5:             Inject($m$)
6:         case received source $s$'s message $m$ from neighbor $n$
7:             Receive_Message($s$,$m$,$n$)
8:         case link-level protocol is ready to send message to neighbor $n$
9:             Send_Message($n$)
```

Similar to other dissemination approaches, Priority Flooding is an event-driven system. When an event occurs, it is processed by the protocol's event handler and the appropriate function is invoked. The first two events occur when a node receives a message from a connected source client or a neighbor in the network topology. The last event in the handler, which invokes the Send_Message function, occurs when a node has a message to send to a neighbor and the corresponding lower-level link protocol (see Section 4) has room to accept a new message.

Initialize() - initializes data structures

```
1: Initialize dynamic data structures
2: Sequence_Number = 1
3: for all $s \in$ sources
4:     if $s \neq me$                                   // If $s$ is not this node
5:         Incarnation[$s$] = 0
6:     else                                             // $s$ is this node
7:         Incarnation[$s$] = get_time_of_day().sec
8: return
```

The Initialize function, which is invoked at the start of the event handler, is responsible for setting up a node's state. Each node uses the information in the maximal topology, i.e. its degree and maximum number of source nodes, to dynamically allocate the appropriate size for the relevant node-common and link-specific data structures. In addition, the node sets the next sequence number to assign to 1, records the current time in seconds as its own incarnation, and sets the incarnation for all other potential source nodes to 0.

Inject(m) - inject message m into the network

```
1: $m$.sequence_number = Sequence_Number
2: Sequence_Number = Sequence_Number + 1
3: $m$.incarnation = Incarnation[$me$]
4: Assign $m$ a priority level                          // Chosen by client or this source node
5: Sign_Message($m$)                                    // Digitally sign message $m$
6: Receive_Message($m$,$me$,$\emptyset$)                // Receive $m$ from myself to trigger function below
7: return
```

The Inject function takes plain message payloads that a node receives from a connected client and prepares them for dissemination by appending Priority Flooding headers. Each message is stamped with the appropriate sequence number, incarnation, and priority level. In addition, the message payload and header information is digitally signed by this node so that subsequent nodes in the network can authenticate the message. Finally, the Receive_Message function is invoked to place the new message into the node's data structures and to forward the message to the appropriate neighbors.

Receive_Message(s,m,n) - receive source s's message m from neighbor n

```
1: if Validate_Message($m$) = False                     // Validate size, headers & signature
2:     return
3: if $m$.incarnation < Incarnation[$s$]                // $m$ is from old incarnation
4:     return
5: else
6:     if $m$.incarnation > Incarnation[$s$]            // $m$ is from new incarnation
7:         Incarnation[$s$] = $m$.incarnation           // Update stored incarnation for $s$
8:
9: if $m \in$ hash_table[$s$]                           // $m$ is a duplicate message
10:     if $m$ is queued toward $n$                     // $m$ is in $n$'s priority queue
11:         Remove $m$ from $n$'s priority queue
12: else                                                // $m$ is unique, first time receiving it
13:     Insert $m$ into hash_table[$s$]                 // Common storage of messages for all links
14:     for all neighbors $i$                           // For each neighbor $n$
15:         if $i \neq n$ and $i \neq s$                // Not source or forwarder
16:             Insert $m$ into $i$'s priority queue
17:             $i$.total_messages = $i$.total_messages +1
18:             $i$.messages[$s$] = $i$.messages[$s$] +1
19:             Update $i$.min_priority[$s$] and $i$.max_priority[$s$]
20:             if $s \notin i$'s fair sending queue
21:                 Insert $s$ into back of $i$'s urgent fair sending queue
22:             repeat
23:                 Send_Message($i$)                   // Try to send messages to $i$
24:             until Sending_Error                     // Until blocked
25:             if $i$.total_messages > Capacity        // Link memory is full
26:                 Find source $h$ with most messages queued toward $i$
27:                 Drop $h$'s lowest priority, oldest message from queue
28: return
```

In addition to being invoked from the Inject function, Receive_Message is triggered when a node receives a message from one of its neighbors. First, the function checks the validity of the message, verifying the signature on the message, the message size, and the header contents. Next, the node compares the incarnation on the message with the stored incarnation of the originating source node. If the message incarnation is lower than what this node has stored, the message is old and discarded. If the incarnation is higher than what is stored, the node knows that the source has crashed and restarted, and stores the new value.

Next, the node determines whether or not this message was already received by performing a constant-time look-up in the source's hash table. If the message is a duplicate, and if that message is queued to be sent to the neighbor that forwarded the message, then the message can safely be removed from that neighbor's queue. If the message is unique, the node first inserts the message into the source's hash table. Next, the node loops through all of its neighbors in order to insert the message in each link's message priority queue, unless the neighbor is the forwarding neighbor or the originating source of the message. After inserting the message into a link's priority queue, the node updates the link statistics, i.e., the total number of messages stored on the link, the number of messages stored from the message's source on the link, and potentially the maximum and minimum priority level stored for the source on the link. If the new message is now the only one stored for the source, and the source is not in the fair sending queue, the source is added to the back of the urgent queue. Once the message is queued, the node knows that it has at least one message to forward to its neighbors. The node repeatedly invokes the Send_Message function to forward messages until either the node runs out of queued messages or the link-level protocol runs out of room to accept new messages. If the Send_Message function fails the first time it is invoked, there is a possibility that the link is storing more messages than allowed by the link capacity. In this case, the node finds the source with the most messages stored on the link, and drops that source's oldest, lowest priority message.

Send_Message(n) - send message to neighbor n

```
 1: if $n$'s fair sending queue is empty
 2:     return Sending_Error
                                        // Find the source next in line to send a message
 3: if $n$'s urgent fair sending queue is not empty
 4:     Select source $s$ that is head of urgent fair sending queue
 5: else                                // $n$'s normal fair sending queue is not empty
 6:     Select source $s$ that is head of normal fair sending queue
 7:
 8: Transmit oldest, highest priority message stored for $s$ in $n$'s priority queue
 9: Move source $s$ to back of $n$'s normal fair sending queue
10: $n$.total_messages = $n$.total_messages $-1$
11: $n$.messages[$s$] = $n$.messages[$s$] $-1$
12: Update $n$.min_priority[$s$] and $n$.max_priority[$s$]
13: return Sending_Success
```

The Send_Message function is invoked with a target neighbor and is responsible for transmitting the oldest, highest priority message from the source that is next in line to send. If the fair sending queue is empty, the node has no messages to forward to the target neighbor. If the queue is not empty, the next source in line to send is the head of the urgent queue, or if that is empty, the head of the normal queue. Once a source is selected, the link forwards that source's oldest, highest priority message that is stored in the message priority queue. After sending the message, the selected source is moved to the back of the normal queue, and the link statistics (total messages, number of messages stored for the source, and maximum and minimum priority level stored for the source) are updated to reflect the forwarded message.

3.3 Trade-offs

The Priority Flooding protocol contains several design choices and system parameters (Section 3.2.3) that offer different trade-offs for a network administrator. Each of these trade-offs affects system properties, such as performance, memory footprint, or design complexity.

Sequence Numbers with Incarnations

Usually, when a source's next sequence number to assign reaches the maximum value, the field wraps around back to 1. However, naively resetting the sequence number will cause subsequent messages to be seen as duplicates and dropped at intermediate nodes. In order to avoid this scenario, the source needs to tell the other nodes in the network that it is resetting the sequence number. One option is to have the source flood a special message through the network that indicates a sequence number reset. The benefit of this approach is that the size of the sequence number field on the message can kept quite small. However, adding another message type to the protocol adds complexity, and the network must ensure that the special message is delivered before messages that use the reset sequence numbers.

Another approach to solve the sequence number wrap-around problem is to use an incarnation. In this case, messages in the network are uniquely identified by three values: the originating source of the message, the source's incarnation number, and a sequence number. When the sequence number reaches the maximum value, a source node can simply update its incarnation and start sequencing messages from 1 again. The higher incarnation implicitly tells intermediate nodes in the network that messages from that source are not duplicates. The advantage of this approach is its simplicity; neither new message types nor new delivery semantics are required. However, the disadvantage is the overhead that the incarnation and sequence number fields use in each message's headers. For our implementation, we choose to use this second approach in order to keep the protocol design simple.

Hash Table for Message Storage

The type of data structure that a node uses to store received messages determines the efficiency of operations. In Priority Flooding, nodes continuously insert, look-up, and delete messages from the data structure. A commonly used data structure that provides $O(1)$ inserts, look-ups, and deletes is a hash table. For this reason, we choose to use hash tables to store received messages. However, a disadvantage of hash tables is that they do not provide an efficient way to sort entries for iteration. For example, consider a scenario where a network administrator wants to find the X lowest sequence-numbered messages a node has stored for a particular source. In the worst case, all of the entries in the hash table must searched, resulting in $O(N)$ operations.

On the other hand, a data structure that supports iterative operations extremely well is a skip list. Entries in a skip list are maintained in sorted order. Performing the same task only costs on average $O(logN) + X$ operations. However, skip lists perform worse than hash tables on operations such as inserts, look-ups, and deletes, costing $O(logN)$ on average. It is up to the network administrator to decide which data structure to use.

Link Capacity

The link capacity determines the number of messages that each link's message priority queue will store, across all source nodes in the network. Naturally, a larger link capacity translates to a larger memory footprint of the Priority Flooding node and links. However, the link capacity can also affect the number of messages that get delivered and the latency experienced by the messages.

To easily illustrate this point, consider a scenario where every source node in the network sends all messages with the same priority level. On a link, if the number of incoming messages exceeds the number of outgoing messages, the message priority queue will fill up. When the link is prompted to forward a message for a source, it selects the oldest, highest priority message it has stored. If the link capacity is small, this message will be a relatively new message that has only waited in the queue for a short time. If instead the link capacity is very large, the forwarded message will be a much older message that has waited in the queue for a long time. Since latency is measured end-to-end between source and destination, the average latency of messages with small link capacity is much less than that with large link capacity because messages do not spend as much time in link queues. However, note that with small link capacity, it is likely that not as many messages get delivered over time, since the link queues simply cannot store as many messages for links to forward.

Type of Cryptographic Authentication

There are several different types of cryptographic signatures that can be used for the public-key authentication in Priority Flooding. Two popular options are RSA signatures [27] or elliptic curve signatures (ECDSA) [28]. The main advantage of using RSA signatures is that verification takes a relatively small amount of time. The disadvantages are that generating signatures is expensive and the size of each signature is on the order of hundreds of bytes. In contrast, ECDSA signatures are much smaller in size (tens of bytes) and the time to generate a signature is less than in RSA. However, verifying an ECDSA signature is more expensive than in RSA.

For our protocol, only the source of each message must generate a signature, but all intermediate nodes that receive the message must verify it. With multiple sources sending simultaneously, we choose to use RSA signatures in order to keep the expensive operations on the edge of the network. However, if the network is bandwidth-bound, using ECDSA signatures is a better choice because it reduces the signature size overhead on each message.

3.4 Proof of Correctness

In order to show that Priority Flooding will succeed under our network model, we prove the two desired guarantees from Section 2.4.

Theorem 1. *If there exists a correct path from a correct source node S to a correct destination node D, then the rate at which S can send messages to D is no less than $\frac{1}{n}$ times the bandwidth of that path.*

Proof. On each link in this correct path, a correct node A is sending to another correct node B. A determines which message to send next by using the two queues from the fair sending queues data structure, the urgent queue and the normal queue. The combined size of these queues is no more than n, since there are only n total sources in the system.

A source in the fair sending queues that sends a message must wait at least one full cycle before being chosen to send another message. Even if the source is inserted into the urgent queue, it must wait in the normal queue before either sending again or leaving the two queues. The psuedocode for function Send_Message() specifies that a source chosen from the urgent queue must be placed at the back of the normal queue, even if it did not send a message.

Therefore, a given source in the two queues will come to the front of the two queues at least once every n messages. Then, if this source always has messages to send, 1 out of every n messages will be from this source, implying that it receives $\frac{1}{n}$ of the bandwidth on this link.

If a given source receives $\frac{1}{n}$ of the bandwidth of each link on the path, and since the bandwidth of a path is defined to be the minimal bandwidth over all links in the path, then that given source also receives $\frac{1}{n}$ of the bandwidth of the path.

□

Theorem 2. *If the network has no highest-priority message from a source S, then if source S introduces a single highest-priority message into the network, the destination D will receive that message within some time t, where t is no greater than the path latency of the shortest correct path.*

Proof. The network has no highest-priority message from source S when no node in the network has any messages from source S stored at the highest priority level in its message priority queue. Each node in the shortest correct path is a correct node, and follows the specified protocol, sending messages stored in the message priority queue from highest to lowest priority.

When a correct node A receives a highest-priority message m from source S, m is stored in the message priority queue for S at the highest priority level. Then, the next time A forwards a message from source S, it chooses a message from the highest priority level, since there is at least one message stored at this level. If m is the only message stored at the highest priority level, it must be the message that is forwarded. This case is guaranteed to occur when the network has no highest-priority message stored from a source S other than message m.

Consider two correct nodes A and B that share a link on the shortest correct path, neither of which have a highest-priority message stored from source S. Once A receives a highest-priority message m from source S, the next message from S that A forwards to B is m. Thus, the time between when A receives m and B receives m is no worse than the sum of the worst case queueing delay and the link propagation delay. By definition, this is the link latency between A and B.

Since the introduced message is the highest-priority message from source S at every node in the shortest correct path, then the message experiences no worse than the link latency on every link in the path. By definition, the sum of the link latencies in the shortest correct path is the path latency of the shortest correct path. Therefore, the destination D will receive the highest-priority message from source S within some time t, where t is no greater than the path latency of the shortest correct path.

□

Chapter 4

Reliable Intrusion-Tolerant Link

Although Priority Flooding is a complete intrusion-tolerant dissemination algorithm, deploying it real-world settings raise several practical barriers. For example, Priority Flooding nodes do not natively get feedback from their neighbors, which is required to provide the retransmission of dropped packets due to packet loss on underlying network links. In order to ensure that Priority Flooding (and Priority Multicast) operates correctly in a real-world setting with potentially malicious components, we implement a Reliable Intrusion-Tolerant Link protocol to manage every connection in the network. The Reliable Intrusion-Tolerant Link protocol provides reliable hop-by-hop communication, maintains cryptographic integrity between neighbors, and prevents a potentially malicious neighbor from consuming a correct node's resources.

4.1 High-Level Description

The Reliable Intrusion-Tolerant Link protocol is a TCP-fair hop-by-hop message passing protocol that provides reliable communication in the presence of a potentially malicious endpoint. The protocol is designed to be a generic solution that any dissemination algorithm can use if it requires intrusion-tolerant communication on each link. When a dissemination algorithm needs to send a message on one of its links, the algorithm can invoke the link protocol's forward data function rather than calling a kernel's native TCP or UDP send function. Similarly, when the link protocol receives a message, it simply passes the message to the dissemination algorithm.

To achieve reliable communication, the Reliable Intrusion-Tolerant Link uses a variation of the standard Selective Repeat ARQ protocol. Each endpoint of a link maintains an outgoing message buffer, which stores and sends messages on the link. For received messages, each endpoint maintains an incoming message buffer that generates cumulative acknowledgements for received messages and negative acknowledgements for messages that the receiver is missing. An endpoint can only advance its outgoing buffer window when it receives a cumulative acknowledgement, which states that the receiving endpoint has successfully received every message up to and including the one explicitly specified in the acknowledgement. Message loss is recovered using the negative acknowledgments. Whenever an endpoint receives a negative acknowledgement, it retransmits the specified message.

Similar to the cryptographic authentication that is required in Priority Flooding to prevent spoofing and tampering, the Reliable Intrusion-Tolerant Link protocol also requires integrity checks on messages sent on a link. One option is to use digital signatures based on public-key cryptography to authenticate messages on each link, similar to the way that nodes authenticate messages traveling through the network in Priority Flooding. However, this option incurs significant computational overhead. For a typical message arriving at a node, the receiving link needs to validate the link-level signature, the node needs to validate the network-wide signature, and finally the outgoing link needs to generate a new link-level signature.

Rather than using digital signatures to provide the authentication, the Reliable Intrusion-Tolerant Link protocol uses Hash-based message authentication codes (HMAC) [29] based on symmetric-key cryptography. The main advantage of using HMACs is that they are much less computationally intensive compared to most public-key cryptography schemes. As a result, links spend less time on cryptographic authentication and more time on receiving and forwarding actual data. In order to establish a shared secret key between the two endpoints of a link, the link protocol performs an authenticated Diffie-Hellman [30] key exchange, leveraging from the fact that each endpoint knows the other's public key. With an established shared secret key, links can use HMACs to provide integrity for messages sent to and received by direct neighbors, ensuring that only messages originating at the link's endpoints are accepted.

In order to be intrusion-tolerant, the link protocol must ensure that an action taken by a malicious endpoint does not affect the endpoint on the other side of the link. For TCP-fair protocols, one attack in particular involves the use of optimistic acknowledgments to cause denial of service [31]. An optimistic acknowledgement is an acknowledgment for a message that an endpoint has not yet received. A malicious receiving endpoint can craft optimistic acknowledgements to acknowledge legitimate packets that a sender already injected into the network. As a result, the sender thinks that the other endpoint is receiving packets faster than the actual transmission rate and increases the sending rate accordingly. The sender can potentially exhaust all of its network bandwidth, resulting in a denial of service.

To prevent the optimistic acknowledgement attack, the Reliable Intrusion-Tolerant Link protocol forces an endpoint to prove that it actually received all messages that it tries to acknowledge. Every message that is sent on a link is assigned a cryptographic nonce. The only way for an endpoint to acknowledge a message is by sending that message's corresponding nonce back to the sender along with the acknowledgment. In order to avoid having a receiver explicitly send a nonce for each received message, the link protocol maintains a nonce digest that is updated whenever the next message is received. An endpoint can prove that it received all messages up to and including sequence i by sending the nonce digest with the i^{th} nonce applied in a cumulative acknowledgement.

4.2 Detailed Description

4.2.1 Data Structures

Each link endpoint maintains the following data structures.

Outgoing Message Buffer

The outgoing message buffer is an array of packet objects that store the messages that Priority Flooding requests to be sent on a link. The size of the array is equal to the administrator-defined window size. Each active packet object stores a pointer to the message to send, the length of the message, a timestamp that indicates when the packet was given to the link, the nonce that is generated for the message, a *NACK* flag which indicates if this packet needs to be retransmitted, and a timestamp that indicates when the packet was last sent. The outgoing buffer contains a head and tail sequence number. The head sequence number is the highest unused sequence number to assign to the next accepted message. The tail sequence number refers to the oldest sequence number of the packet that the receiving endpoint has yet to cumulatively acknowledge.

The link protocol accepts messages from Priority Flooding as long as the outgoing message buffer is not full, or when the difference between the head and tail sequence numbers is less than the window size. When a message is accepted, it is assigned the head sequence number, placed in the array at the corresponding packet object index (head sequence modulo window size), and the head sequence number is incremented. When the receiving endpoint sends back a cumulative acknowledgement that is greater than or equal to the tail sequence number, the tail is moved up accordingly and more space is made available in the outgoing window.

In order to prevent a malicious endpoint from optimistically acknowledging, each cumulative acknowledgment must contain the correct nonce digest for all of the messages up to that point, starting from sequence number 1 and ending at the acknowledgement sequence number. When the receiving endpoint sends a negative acknowledgment, the link marks the NACK flag of the packet with that sequence number to signal that the packet should be retransmitted. In order to prevent a malicious endpoint from consuming resources by continuously sending negative acknowledgements, messages are retransmitted no faster than a specified rate. Each time a packet is sent, the time is recorded in the packet object. Then, a packet is only retransmitted if it is NACK'd again and the administrator-defined timeout has elapsed since that packet was last sent.

Incoming Message Buffer

The incoming message buffer is an array of cells that mirrors the outgoing message buffer on the other side of the link. The size of the array is also equal to the window size. Each cell stores a single packet and its corresponding cryptographic nonce. As with the outgoing buffer, the incoming message buffer contains a head and tail sequence number. The head sequence number is the sequence number of the highest received packet plus one, and the tail sequence number refers to the lowest sequence number that cannot yet be cumulatively acknowledged.

When an endpoint receives a message from the link, the packet is stored in the incoming message array, indexed by its sequence number modulo the window size. If the sequence number of this packet matches the head sequence number, it is the next expected packet and is stored accordingly. If the packet's sequence number is greater than the head, the packet is stored and a negative acknowledgement is created for the packets in the gap. If the packet's sequence number is below the head, it is either a duplicate packet (and discarded) or it fills in a gap in the incoming array where a packet had previously been marked as missing. Whenever the packet corresponding to the tail sequence number is received, the nonce digest is updated, the tail is incremented, and a new cumulative acknowledgement can be generated.

Nonce Digest

In order to prevent a receiving endpoint from optimistically acknowledging packets that it has not actually received and from consuming the sending endpoint's resources, each sent packet includes a cryptographic nonce. When a receiving endpoint wants to send a cumulative acknowledgement, it must also send the nonce digest corresponding with the sequence number in the acknowledgement. The nonce digest for sequence number X is the bitwise exclusive or of the nonces on all of the packets starting at sequence 1 and ending at sequence X. For example, in order to acknowledge packet with sequence number 3, the receiver must calculate nonce_1 XOR nonce_2 XOR nonce_3 and include it on the acknowledgement.

For outgoing messages, the link stores the digest that corresponds with each packet between the head and tail sequence numbers: the packets that have yet to be acknowledged. When the receiver acknowledges one of these packets, the sender can quickly check the nonce digest on the acknowledgment with the precomputed digest that is stored for the packet. For incoming messages, the link stores the digest of all the contiguously received nonces, i.e., the nonces up to and including the nonce associated with sequence number one less than the tail sequence number. Only a single digest is necessary on the receiving side since cumulative acknowledgements cannot be generated for packets beyond the tail. Once the packet corresponding to the tail sequence number is received, the digest is updated with that packet's nonce and the tail is incremented.

4.2.2 Message Types

Data Message

A Reliable Intrusion-Tolerant Link data message contains the actual data to be sent on the link passed down by Priority Flooding, as well as additional protocol-specific information appended to the end of the message. This information includes the sequence number of that message on that link and the random nonce for that message. It also includes an all-received-up-to value, specifying that the endpoint sending this data message has received all the messages sent to it on this link up to and including that sequence number. To prove that it actually received all those messages, it includes the digest of all the nonces of all those packets as well. Finally, the message includes a list of negative acknowledgements, or sequence numbers that have not yet been received by this participant and need to be resent. The data message is protected by an HMAC using the shared secret key.

Stand-Alone Acknowledgement

A stand-alone acknowledgement contains up-to-date link-level header information, but does not actually contain a message payload. This allows the protocol to send acknowledgement information even when no data is being sent in that direction on the link.

4.2.3 Pseudocode

Link_Event_Handler() - processes link events

```
while true
    switch event
        case received message $m$ from Priority Flooding
            Forward($m$)
        case received message $m$ on the link
            Link_Receive($m$)
        case timeout to send stand-alone acknowledgement
            SAA_Timeout()
```

The Reliable Intrusion-Tolerant Data link is also event-driven. When an event occurs, it is processed by the link's event handler and the appropriate function is invoked. The first two events occur when the link receives a message from either the Priority Flooding dissemination or the other endpoint on the link. The last event occurs in response to a triggered timeout. In this case, the link has a new cumulative acknowledgement to send, but has no data messages to piggyback the acknowledgment on, and therefore sends a stand-alone acknowledgement packet.

Forward(m) - accepts and sends message m from Priority Flooding

```
1: if outgoing_head − outgoing_tail ≥ window_size                // no room for message m
2:     return
3: sequence = outgoing_head                         // Get the sequence number of the new packet
4: outgoing_head = outgoing_head + 1                                       // Advance the head
5: p = outgoing_buff[sequence % window_size]          // Point p at corresponding array index
6: p.sequence_number = sequence
7: p.payload = m
8: p.length = m.length
9: p.nonce = Generate_Random_Number()
10: update outgoing_nonce_digest[sequence % window_size]
11: p.last_sent = now
12: p.aru = incoming_tail − 1
13: p.aru_nonce = incoming_nonce_digest
14: Append_NACKs(p)
15: p.hmac = Generate_HMAC(p)                                 // create the HMAC for packet p
16: Link_Send(p)                                                 // Send packet p on the link
17: return
```

The Forward function accepts a new message from Priority Flooding, creates a packet for that message, and sends the packet on the link. First, the function checks that it has room to accept new messages in the window. If there is room, the message is encapsulated into a link-level data packet and stored in the outgoing buffer. The packet is given the next sequence number, a reference to the message, the length of the message, and a cryptographic nonce. In addition, the packet is given the all-received-up-to sequence number and the corresponding nonce digest for that sequence number. Next, an HMAC is generated for the information in the packet and stored as well. Finally, the packet is sent on the link to the other endpoint.

Link_Receive(p) - receives packet p on the link

```
1: if Validate_HMAC($p$) = False                                  // Checks the HMAC on the packet
2:     return
3: if $p$.sequence_number < incoming_tail or $p$.sequence_number > incoming_tail + window_size
4:     return                                                 // Verifies the packet is within bounds
5:
6: if outgoing_tail $\leq$ $p$.aru < outgoing_head
7:     if $p$.aru_nonce = outgoing_nonce_digest[$p$.sequence_number % window_size]
8:         outgoing_tail = $p$.aru + 1                                    // Move up outgoing tail
9:
10: if $p$ contains NACKs                              // packet $p$ contains negative acknowledgements
11:     for all $n \in$ NACKs
12:         if outgoing_buff[$n$ % window_size].last_sent + NACK_timeout < now
13:             Retransmit($n$)
14:             outgoing_buff[$n$ % window_size].last_sent = now
15:
16: if incoming_buff[$p$.sequence_number % window_size].message = $\emptyset$
17:     incoming_buff[$p$.sequence_number % window_size].message = $p$.payload
18:     incoming_buff[$p$.sequence_number % window_size].nonce = $p$.nonce
19:     if $p$.sequence_number > incoming_head      // $p$ is not the next expected sequence number
20:         for $i \in$ [incoming_head, $p$.sequence_number $-$ 1]
21:             Generate_NACK($i$)
22:     while incoming_buff[incoming_tail % window_size].message $\neq \emptyset$  // we can advance the tail
23:         update incoming_nonce_digest with nonce at incoming_tail
24:         incoming_tail = incoming_tail + 1
25:     incoming_head = $p$.sequence_number + 1
26:     Clear_NACK($p$.sequence_number)
27:     Deliver($p$.payload, $p$.length)             // Gives the message to Priority Flooding algorithm
28: return
```

The Link_Receive function contains the majority of the logic in the Reliable Intrusion-Tolerant Link protocol. First, the function performs sanity checking on the message, ensuring that the HMAC verifies and that the sequence number of the received packet is within the outgoing message window. Next, the all-received-up-to sequence number is checked. If the aru is within bounds, the nonce digest of the aru is verified and the outgoing_tail can be moved up accordingly. Next, the link checks if the packet has any negative acknowledgements. For all of the NACKs, the corresponding packet is retransmitted only if the required timeout has expired since the message was last sent. This prevents the other endpoint of the link from consuming this endpoint's bandwidth. Finally, the message in the packet is processed. If the packet is not a duplicate, the message payload and the nonce are stored at the correct index in the incoming message array. If the packet's sequence number is above the head, it is not the next packet that is expected and a gap is created. A negative acknowledgement is generated for all the missing packets. If instead the packet's sequence number filled in the missing packet at the incoming tail, the incoming nonce digest is updated and the incoming tail is advanced. The incoming head is advanced, any NACKs for this sequence number are removed, and the received message is delivered to the higher-level Priority Flooding dissemination algorithm.

SAA_Timeout() - sends stand-alone acknowledgement

```
1: Enqueue SAA_Timeout()                  // Enqueue this function for the next timeout
2: if received_messages = 0    // number of received messages is 0 since last SAA_Timeout
3:     return
4: Send_SAA()                          // send stand-alone acknowledgment on the link
5: received_messages = 0
6: return
```

The SAA_Timeout function sends a stand-alone acknowledgment when the link has a new cumulative acknowledgment to send but has no data packets to send on which to piggyback the information. First, the function re-enqueues itself, since it should continuously check to see if a stand-alone acknowledgment should be sent. Next, the function checks if any messages have been received since the last time a stand-alone acknowledgment was sent. If no new packets were received, then the link does not have new information to send. If the number of received messages is greater than zero, the link sends a stand-alone acknowledgment, sets the number of received messages to zero, and returns.

Chapter 5

Implementation Considerations

To validate Priority-Based Flooding with Source Fairness and the Reliable Intrusion-Tolerant Link, we implement both protocols for deployment on network topologies in the Spines overlay networking framework [32]. In this chapter, we describe the implementation considerations for both protocols in Spines and discuss the benefit of source-defined expiration time in Priority Flooding.

5.1 Source-Defined Message Expiration

In the Priority Flooding design, received messages are stored at each node in the corresponding source's hash table. Even when all of a node's neighbors no longer need the message, the metadata of the message must continue to be stored to prevent replay attacks from a potentially malicious neighbor. In real deployment, this behavior would require an unbounded amount of memory at each node if Priority Flooding continues to run indefinitely. In order to limit the amount of memory, each message can be assigned an expiration time at the source that is covered by the digital signature. As a result, each node only needs to store the metadata for messages that have yet to expire. If a malicious node replays an unexpired message, a correct node will still recognize the message as a duplicate from the stored metadata. If the malicious node instead replays an expired message, the correct node will see that the message is expired and simply drop the message rather than store it. Since the expiration time is source-defined and protected by the digital signature, a malicious node cannot change the expiration time to try and trick a correct node into thinking it is a new message for which no metadata is stored. With an expiration time on each message, Priority Flooding can now garbage collect metadata periodically by iterating through the hash tables and looking for expired messages.

In addition to the memory benefits, source-defined expiration time provides sources with another level of customization for their messages. Similar to how sources assign different priority levels to influence which messages get delivered, sources can set varying expiration times, depending on how relevant each message is over time, to influence the timeliness of messages. For example, a message that is only important to the destination for a short period of time should be assigned a low expiration time that reflects the short period. If that message is not forwarded within its relevant period, nodes can drop the message and move on to forwarding messages that are still relevant to the destination. Therefore, using appropriate expiration times increases the timeliness of sources' relevant messages. Note that a message that is always meaningful to the destination should be assigned a very high or even indefinite expiration time, which forces nodes to retain the message until it is forwarded or dropped due to memory contention.

Using source-defined expiration times requires that all of the correct nodes in the network have a basic level of clock synchronization. Each message is stamped with the source's current time, which acts as the origin time of the message, before being injected into the network. To test if a message is expired, other nodes in the network compare their current time with the message's origin time plus the expiration time. If a node's current time is greater, the message has expired. Without clock synchronization, nodes may not correctly distinguish between expired and unexpired messages. In Priority Flooding, we expect that typical expiration times of messages are on the order of tens to hundreds of seconds. At this granularity, most modern systems can achieve a sufficient level of synchronization using standard services, such as *ntp* [33].

5.1.1 Updated Pseudocode

Including source-defined expiration times in Priority Flooding requires some minor changes in the pseudocode from Section 3.2.4. Next, we provide the updated portions of the pseudocode. For convenience, the line numbers that contain expiration-time changes are highlighted in red.

Garbage_Collect() - periodic garbage collection function

```
1: for all Source $s$'s message storage $buf$
2:     for all messages $m$ in $buf$
3:         if $m$ is expired
4:             Remove $m$ from any neighbor's data structures
5:             Delete $m$ from $buf$
6: Enqueue garbage collection function
7: return
```

The Garbage_Collect function is an entirely new function that uses the source-defined expiration time to periodically garbage collect the memory of expired messages. Every timeout, the function iterates through all of the source hash tables at a node and deletes the message and metadata associated with any expired messages. After the cleanup, the function re-enqueues itself to be called at the next timeout.

Initialize() - initializes data structures

```
1: Initialize dynamic data structures
2: Sequence_Number = 1
3: for all $s \in$ sources
4:     if $s \neq me$                                  // If $s$ is not this node
5:         Incarnation[$s$] = 0
6:     else                                            // $s$ is this node
7:         Incarnation[$s$] = get_time_of_day
8: Enqueue Garbage_Collect()
9: return
```

The Initialize function is exactly the same as before, except that it queues up the first invocation of the Garbage_Collect function.

Inject(m) - inject message m into the network

```
1: $m$.sequence_number = Sequence_Number
2: $m$.incarnation = Incarnation[$me$]
3: Assign $m$ a priority level and expiration time     // Chosen by client or this source node
4: Sign_Message($m$)                                   // Digitally sign message $m$
5: Receive_Message($m$,$me$,$\emptyset$)               // Receive $m$ from myself to trigger function below
6: return
```

The only change in the Inject function is that the source now assigns each of its messages a source-defined priority level and expiration time. The expiration time is placed on the message and covered by the digital signature.

Receive_Message(s,m,n) - receive source s's message m from neighbor n

```
    <Lines 1-8 from Receive_Message function in Section 3.2.4>
 9: if $m \in$ hash_table[$s$]                         // $m$ is a duplicate message
10:     if $m$ is queued toward $n$                    // $m$ is in $n$'s priority queue
11:         Remove $m$ from $n$'s priority queue
12: else                                               // $m$ is unique, first time receiving it
13:     if $m$ is expired                              // do not forward expired messages
14:         return
15:     Insert $m$ into hash_table[$s$]                // Common storage of messages for all links
    <Lines 14-28 from Receive_Message function in Section 3.2.4>
```

Similar to the other functions, the Receive function only has a minor addition. Due to the length of the function, we only include an excerpt of the code. In the new Receive function, a node that does not have the metadata for a message must first check if the message is expired before deciding to store and forward the message. This addition prevents a malicious neighbor from successfully launching a replay attack with expired messages.

5.1.2 New System Parameters and Trade-offs

The addition of source-defined message expiration times and a garbage collection function creates new system parameters and trade-offs.

Garbage Collection

The period between two consecutive invocations of the Garbage_Collect function is determined by the *garbage_collection_timeout*. For optimal memory utilization, metadata should be discarded as soon as it expires. Searching the hash tables for expired metadata too often is computationally wasteful, while searching too infrequently results in a waste of memory and high overhead to remove a large volume at once. Thus, the garbage_collection_timeout is left as a system parameter that can tuned as needed by an administrator.

Garbage collecting metadata from the hash tables revisits a trade-off we discussed in Section 3.3. Hash tables provide $O(1)$ cost for operations such as inserts, look-ups, and deletes, but can cost up to $O(N)$ for iterative operations. In the worst case, the garbage collection function will have to iterate through all of a source's messages each invocation, since the messages are not sorted in the hash table. In contrast, a skip list would provide $O(logN)$ cost for both iterative operations and inserts, look-ups, and deletes. For consistency with the design, we choose to use hash tables, but it is up to the administrator to decide which data structure to use.

5.2 The Spines Framework

Spines, publicly available at www.spines.org, is an open source overlay messaging infrastructure that supports unicast, multicast, and anycast communication over dynamic networks [32]. In contrast to other router programming environments, Spines is a flexible solution that supports network reconfiguration on the fly. In addition to supporting standard end-to-end networking, Spines can deploy overlay networks in the Internet to achieve additional services (e.g., overlay multicast, hop-by-hop recovery) that are not normally available in Internet routing.

Setting up and deploying a Spines network is a straightforward process. Virtual router nodes are instantiated on participating computers and virtual edges are specified between these nodes. Each virtual Spines edge actually encapsulates a number of link protocols (e.g., best-effort, reliable), each of which have their own properties in order to fulfill the particular messaging requirements of different applications. With the Spines network in place, packets are automatically routed through the network topology according to one of the many available metrics (e.g., hop distance, latency, or loss rate).

Connecting applications to an instantiated Spines network is simple. Applications that wish to use Spines interact with the Spines library, which presents near-identical API calls to that of the Unix Socket Interface. For example, spines_socket() is used in place of socket() and spines_send() is used in place of send(). As a result, any socket-based application can be adapted to work with Spines.

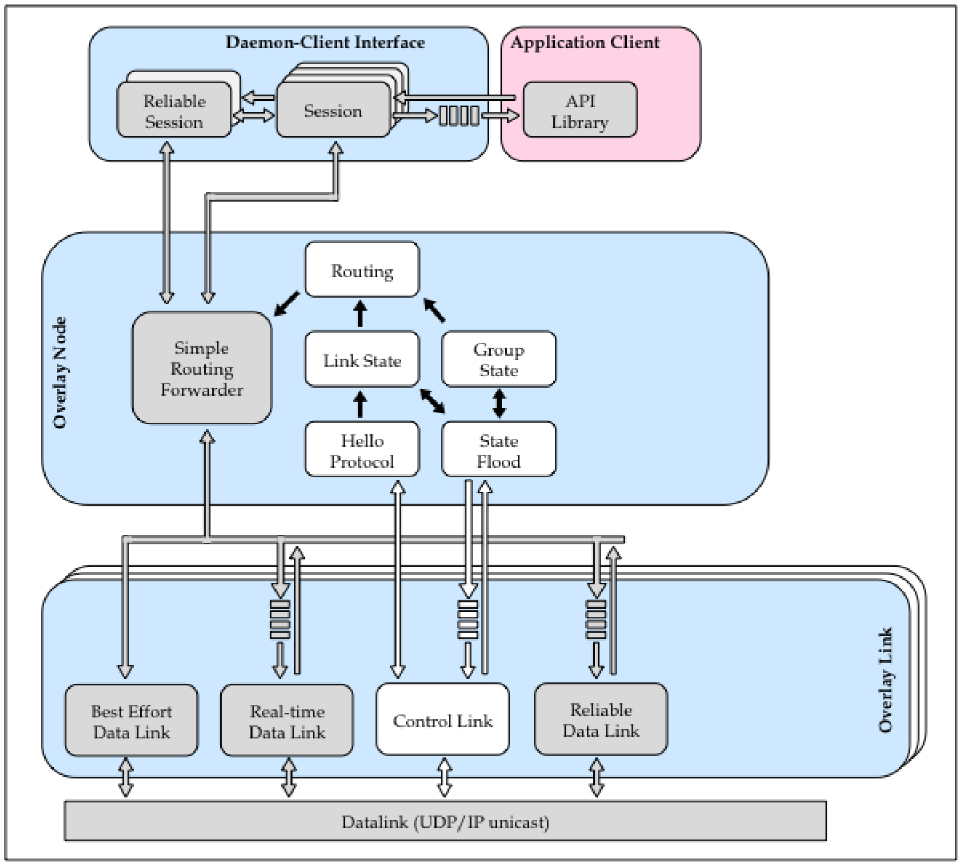


Figure 5.1: Currently-Available Spines Architecture

To date, Spines has been used in both research and production environments for a variety of applications, including Voice over IP, reliable end-to-end communication, and communication between wireless mesh access points, just to name a few. The architecture of the currently available version of Spines can be seen in Figure 5.1.

5.3 Making Spines Intrusion-Tolerant

Spines has achieved tremendous success in benign network environments. However, making it intrusion-tolerant to support Priority Flooding and the Reliable Intrusion-

Tolerant Link requires several architectural changes, since the presence of potentially malicious Spines daemons poses new challenges to the existing framework.

Configuration File

Spines daemons continuously run a discovery protocol to listen for connections from new neighbors in the network. Although discovery is useful in benign environments, accepting ad hoc connections from unauthenticated neighbors in an intrusion-prone environment can lead to a variety of attacks. For example, an adversary can spawn a large number of Spines daemons and have each of them advertise a connection to a target victim daemon. The resources of the victim daemon must be split between all of its neighbors. If the number of malicious daemons is large enough, the amount of resources allocated to each neighbor will be too small for effective communication, essentially resulting in a denial of service to the victim daemon's legitimate neighbors.

To prevent the discovery attack, we introduce a configuration file into the Spines architecture. The configuration file serves as the maximal topology that the offline network administrator distributes to all of the participating nodes. The configuration file lists the identity and IP address of each node, and all of the connections between these nodes. When a Spines daemon starts, it reads the configuration file and creates a local view of the network topology. Daemons only accept messages from their specified direct neighbors; messages from daemons not listed in the configuration file and messages from legitimate daemons that do not have a direct connection are dropped.

Intrusion-Tolerant Priority Flooding

The currently-available version of Spines provides applications with a single dissemination algorithm, shortest-path routing based on one of several metrics, including minimum path latency, minimum path weight, or least number of hops. While shortest-path routing is optimal in terms of bandwidth overhead, routing along a single path in an intrusion-prone environment results in a single point of failure; if any node on the chosen path is compromised, there is no way to guarantee that messages will be delivered to the destination. In addition, the types of attacks we describe in Section 2.5 can have negative effects on metric-based routing schemes, and in some cases can disrupt messaging altogether.

To bypass the negative effects of compromised nodes on metric-based routing, we create a new dissemination algorithm in Spines, Intrusion-Tolerant Priority Flooding. This dissemination algorithm is based on the Priority Flooding design and implementation considerations that are described in Chapter 3 and Chapter 5 respectively. Intrusion-Tolerant Priority Flooding sits next to shortest-path routing in the overlay node logic of the new Spines architecture (see Figure 5.2). Priority Flooding can be configured to use any of the available overlay link protocols, and applications that interact with the new version of Spines can select to deliver their packets with either the shortest-path or Priority Flooding dissemination schemes.

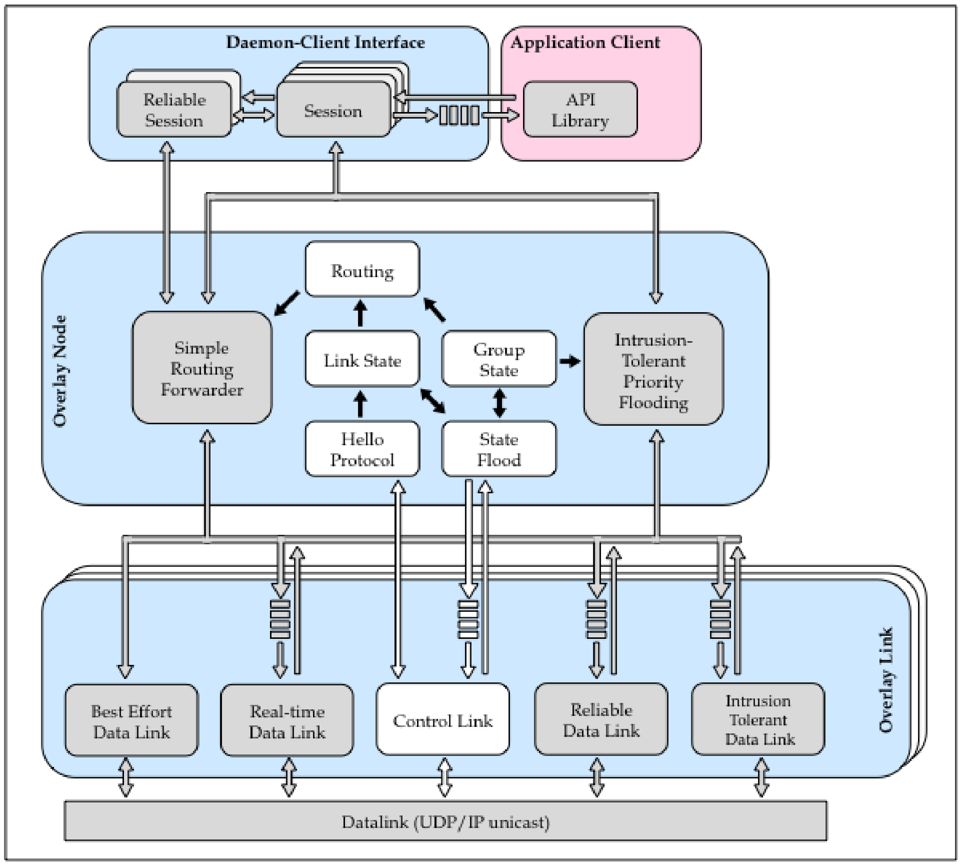


Figure 5.2: Intrusion-Tolerant Spines Architecture

Intrusion-Tolerant Data Link

In order to provide a complete intrusion-tolerant solution, an implementation of the Reliable Intrusion-Tolerant Link protocol is required to supplement the Priority Flooding dissemination algorithm. We implement a new overlay link protocol, the Intrusion-Tolerant Data Link, based on the design and implementation considerations in Chapter 4 and Chapter 5 respectively. The new link protocol delivers packets reliably between two direct neighbors and provides the necessary defenses against a potentially malicious endpoint, such as the use of a cryptographic nonce on each data packet. The Intrusion-Tolerant Data Link protocol sits next to the other overlay link protocols and can be used by any of the dissemination algorithms in Spines (see Figure 5.2). For example, an application using Spines can choose to disseminate messages via shortest-path routing using the Intrusion-Tolerant Data Link on each hop in the path.

Rather than allowing the Intrusion-Tolerant Data Link to consume all of a physical link's bandwidth, a network administrator may want to limit the usable bandwidth to a fraction of the potential total. To support this behavior, we add a new tunable system parameter to the Intrusion-Tolerant Data Link protocol. A network administrator can choose a bandwidth capacity to apply to each overlay link in the Spines network. Each node then maintains source fairness on links using the specified bandwidth limit as the total available bandwidth that can be allocated among active sources.

Cryptographic Authentication

In order to provide cryptographic authentication for both the Priority Flooding and Reliable Intrusion-Tolerant Link protocols, we use the standard OpenSSL library [34]. For Priority Flooding, we use the RSA implementation from OpenSSL for the public-key cryptography. A private-private key pair is generated for each Spines daemon in the network. When a daemon starts up, it reads its private key and the public key of all nodes in the network so that it generate and verify RSA signatures. For the Reliable Intrusion-Tolerant link, we use the Diffie-Hellman and HMAC implementation from OpenSSL. Using the public/private keys, two direct neighbors run an authenticated Diffie-Hellman key exchange to establish a shared secret key. With the secret key established, two direct neighbors generate and verify HMACs (using the SHA-2 hash function implementation) to provide link-level message integrity.

Chapter 6

Evaluation

With the Spines-based implementation of Priority Flooding, we evaluate our protocol to show that it meets the high-availability and real-time demands of cloud monitoring systems, while ensuring fairness among all source nodes in the network. We test Priority Flooding on emulated networks using a LAN cluster, and we demonstrate that the protocol functions as expected and that the desired guarantees are met.

6.1 Environment

The local area network cluster on which we test Priority Flooding is composed of eight servers. Each machine is a Dell PowerEdge R210 II server, with an Intel Xeon E3-1270v2 3.50 GHz processor and with 16GB of memory. All eight machines have Solarflare 5161T 10GbE Ethernet cards and are connected via an Arista 7120T-4S switch, providing a 10 Gigabit Ethernet connection.

In order to emulate a more realistic environment, we instrument a layer in the Reliable Intrusion-Tolerant Link protocol to enforce a bandwidth limit on the traffic sent between two nodes, as mentioned in the Intrusion-Tolerant Data Link portion of Section 5.3. The bandwidth limit is enforced independently on each link in the network. Thus, the total outgoing bandwidth of a node is constrained by the sum of the bandwidth limit on all of that node's outgoing links.

Since Priority Flooding uses cryptographic authentication, each message incurs the overhead of a digital signature. Priority Flooding's protocol-specific headers also contribute overhead to messages. Quantifying this overhead, digital signatures (using 1024-bit RSA keys) and headers add 128 bytes and 64 bytes respectively. In total, this represents 192 bytes of overhead for each message. In our experiments, the message payload size 1000 bytes, resulting in an expected goodput of 84% of the bandwidth limit. To reduce the digital signature overhead on each message, different cryptographic signatures can be used (see section 3.3). In general, using larger message payloads will reduce the percentage of overhead that the signatures and headers contribute to each message.

In all of the experiments, messages are injected into the network by a source client program that connects to an entry node in the network and specifies a target destination node and rate. The source client sends messages to the entry node at the specified rate, and the entry node takes these messages and injects them into the network with the appropriate headers and cryptographic signatures (similar to the Inject function in Section 3.2.4). When a node receives messages for which it is the destination, it forwards the messages to an attached destination client program. It is here at the destination client that the throughput and latency of the flow is measured. Every time the destination client receives 1% of the total messages in the flow, it calculates the rate at which this 1% of traffic arrived as well the latency of the last packet that makes up the 1%. The calculated rate represents the *goodput* of the flow, since it omits messages retransmissions and discounts the overhead due to headers or cryptographic signatures, since they are removed prior to delivering messages to the destination client.

6.2 Experimentation

We test the Priority Flooding implementation on two network topologies. The first topology is simple and shows the properties and guarantees of Priority Flooding in a confined scenario. The second topology shows how Priority Flooding behaves in a more complicated scenario. The second topology is designed to reflect a realistic topology that is used by global cloud providers.

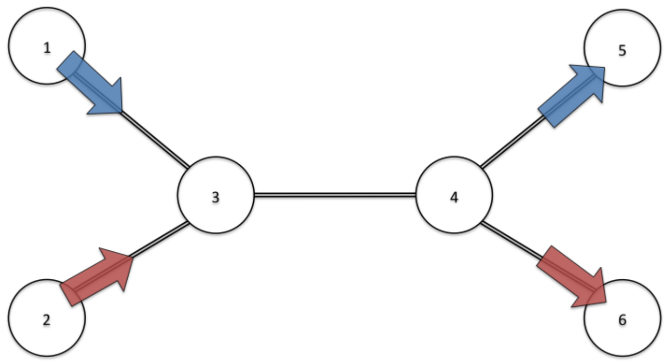


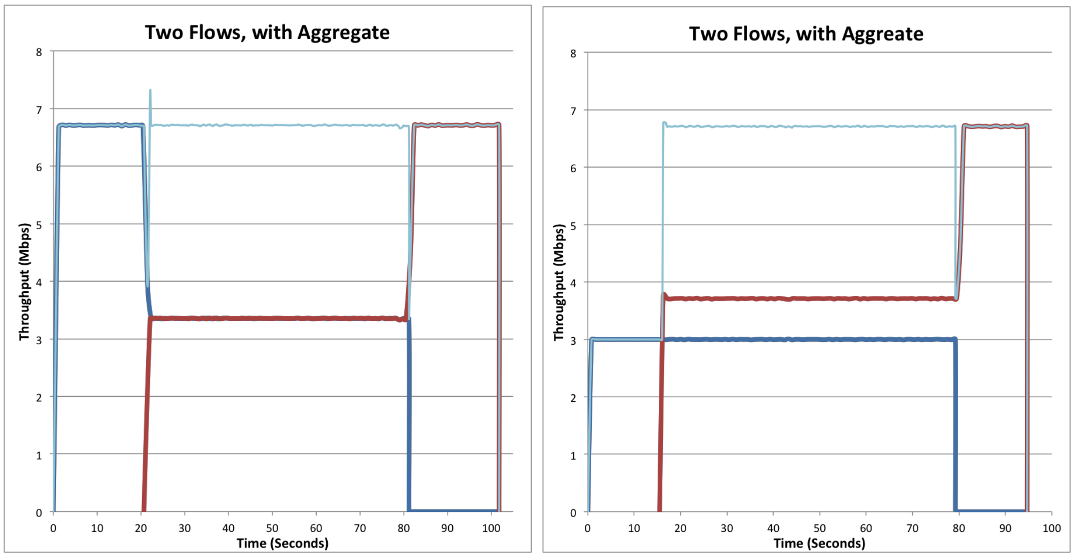
Figure 6.1: Simple bow tie topology for two flows.

6.2.1 Simple Topology

The first experiments use a simple topology with six nodes that are connected to resemble a bow tie (see Figure 6.1). This simple topology ensures that flows going across the network must share the link between nodes 3 and 4. As a result, we can investigate the behavior and fairness of Priority Flooding in a simple case by sending two flows of messages across the network simultaneously. In Figure 6.1, each flow is represented by a pair of colored arrows. The blue flow starts at node 1 and ends at node 5, while the red flow starts at node 2 and ends at node 6. The arrows represent source or destination clients that send messages to an entry node or receive messages from a destination node, respectively. In Figure 6.1, the arrows leaving nodes 1 and 2 represent source clients and the arrows entering nodes 5 and 6 represent destination clients.

With the simple topology and artificial bandwidth limit of 8 Mbps on each link, we run two experiments. In the first experiment, the red and blue flows send at a higher rate than the link's bandwidth limit. Since both flows only travel in the same direction across the link between nodes 3 and 4, that link is the bottleneck in the network. According to the fairness property of Priority Flooding, since this link cannot support the full bandwidth request of either flow, we expect each flow to be given their fair share, or half of the link's bandwidth.

The results of this experiment are shown in Figure 6.2a. Initially, only the blue flow sends messages and can consume all of the resources on the link between nodes 3 and 4. Note that in this case, all of the resources translates to $0.84\% * 8$ Mbps, or 6.71 Mbps. As soon as the red flow starts to send messages, both the blue and red flow receive their fair share of the link's bandwidth, or about 3.35 Mbps. Finally, when the blue flow finishes sending messages, the red flow can consume all of the bandwidth. In the graph, the light-blue line shows the aggregate throughput across both flows over time. We see that the full bandwidth of the network is utilized at all times.



(a) (b)

Figure 6.2: Throughput over time for two flows in the network. (a) shows two equal flows, while (b) shows one flow sending less than its fair share.

In the second experiment, the red flow still sends at a higher rate than the bandwidth limit, but the blue flow sends less than its fair share when there are two active flows in the network. Specifically, the blue flow sends at 3 Mbps, which is less than the 3.35 Mbps that it would be assigned on the link between nodes 3 and 4 with two active flows in the network. According to the fairness property, since the blue flow's bandwidth request is less than its fair share, we expect the blue flow to be completely unaffected by the presence of the red flow, and the red flow to receive all of the remaining bandwidth.

The results of the second experiment are shown in Figure 6.2b. At the start, the blue flow sends at the constant rate of 3 Mbps. Once the red flow starts to send messages, the blue flow is completely unaffected and continues to experience 3 Mbps, while the red flow is able to receive all of the remaining unused bandwidth, or 3.67 Mbps. Finally, when the blue flow finishes sending messages, as we saw in the first experiment, the red flow can consume all of the bandwidth. Again, the light blue-line shows the aggregate throughput across both flows over time. When just the blue flow is sending, the aggregate is 3 Mbps, but once the red flows starts to send, we experience full utilization of the network bandwidth.

The results of both experiments on the simple topology are consistent with Priority Flooding's fairness property, allocating resources dynamically as new flows appear and existing flows disappear. Moreover, Theorem 1 of the desired guarantees (Section 2.4) is achieved, as each flow in the network receives at least $\frac{1}{n}$ of the path bandwidth on the path between source and destination nodes.

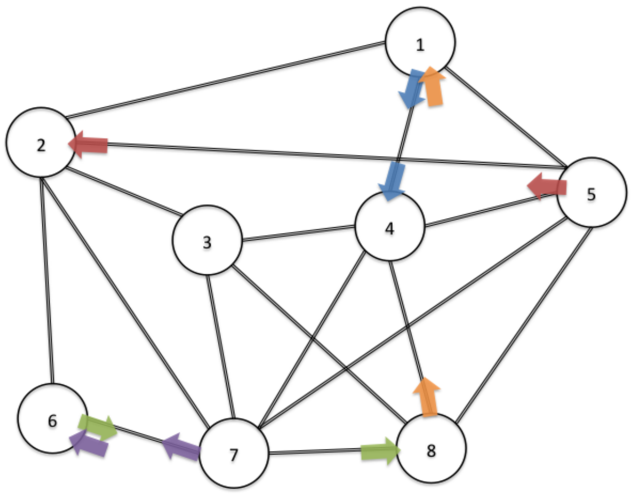


Figure 6.3: Realistic topology for five flows.

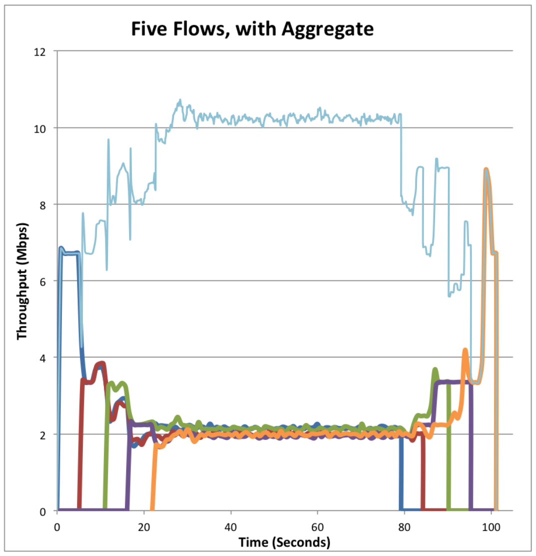
6.2.2 Realistic Topology

The next set of experiments use a more realistic topology (Figure 6.3) that we base on a real global cloud provider's network [35]. In this network topology, there are eight nodes that have multiple links to other nodes in the network. As a result, flows starting at one node have multiple paths to reach the target destination node. In addition to the more complex topology, we emulate real-world latencies on each link, but at the request of the cloud networking provider, these values are not shared. For the experiments on this topology, we deploy five flows, indicated by the colored arrows in Figure 6.3. Specifically, node 1 sends to node 4, node 5 to node 2, node 6 to node 8, node 7 to node 5, and node 8 to node 1. Similar to the experiments on the simple topology, we place an artificial bandwidth limit of 8 Mbps on each link in the network. In order to emulate a realistic monitoring system, each source rotates the priority level that is assigned to each message. In these experiments, the maximum number of priority levels is set to ten. Thus, each source assigns the first message priority 1, the second message priority 2, the tenth message priority 10, the eleventh message priority 1, etc.

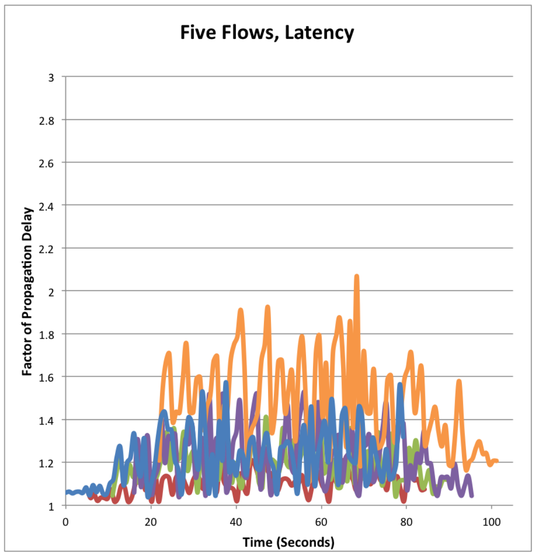
The first experiment on the realistic topology involves all five flows sending at a higher rate than the bandwidth limit on each link in the network. Rather than starting all five flows at once, the flows are started in a staggered manner. This behavior allows us to observe how existing flows are affected by each additional flow that starts. Unlike the experiments on the simple topology using two flows, there is no straightforward prediction of the behavior of each of the five flows. However, we expect the fairness property of Priority Flooding to hold. That is, each flow should receive at least its fair share of bandwidth on every link, which is $\frac{1}{5}*$ 6.67 Mbps = 1.34 Mbps.

The results of this experiment are shown in Figure 6.4. In Figure 6.4a, we see the throughput of each flow over time. Initially, only the blue flow sends messages and can consume all of the bandwidth on each link in the network. As more flows are added, the bandwidth that each flow receives decreases, since there are more flows competing on each link. Once all five flows are actively sending, we see that the fairness property and desired bandwidth guarantees are achieved, as each flow receives at least $\frac{1}{5}$ of the bandwidth limit. Finally, as flows finish sending, the bandwidth that is no longer being used is reallocated to the remaining active flows.

Although the fairness property are met at all times, it is interesting to note that when all five flows are active, each flow receives strictly higher throughput than its fair share, or 1.34 Mbps. The light-blue line, which shows the aggregate throughput across all five flows over time, reinforces this observation. When all five flows are active, the total experienced throughput is about 10.25 Mbps (not 6.67 Mbps), translating to an average of about 2.05 Mbps per flow. The reason for this behavior is a combination of several protocol optimizations and the emulated link latencies. When a node receives a message, it enqueues that message to be sent to all of its neighbors except the one



(a)



(b)

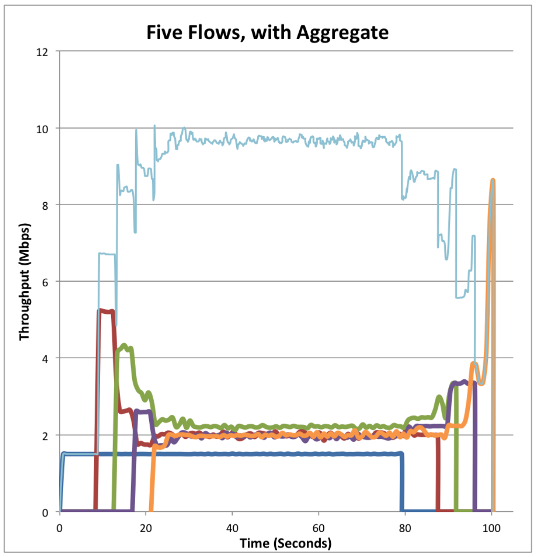
Figure 6.4: Five equal flows in the network. (a) shows throughput over time, while (b) shows factor of propagation delay over time.

that forwarded the message and any neighbors that are the originating source of the message (Line 16 of the Receive_Message function in Section 3.2.4). As a result, links going into a source node will only have four of the flows contending for resources. In addition, by using link latencies, messages will tend to reach some nodes in the network faster than others, resulting in certain flows not crossing every link in both directions. Thus, if there exists a path from a flow's source to destination, where on each link that flow does not contend with all four other flows, that flow will receive more than its fair share (1.34 Mbps).

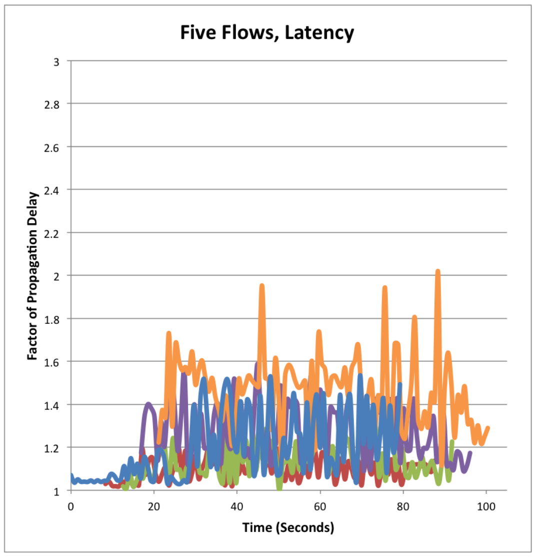
Figure 6.4b shows the latency experienced by each flow over time. Although we cannot show the specific latency values for each flow, at the request of the global cloud provider, we can show the factor of propagation delay. In this metric, a value of 1.0 indicates that the message arrived at the propagation delay of the underlying physical network; a value of 2.0 indicates that the message arrived after twice the propagation delay had elapsed. The graph shows that the maximum and mean factor of propagation delay experienced by the five flows over the whole experiment was about 2.0 and 1.3 respectively. The highest end-to-end propagation delay between two nodes in the realistic topology is on the order of 50 milliseconds. Therefore, a factor of propagation delay of 1.6 results in an average latency on the order of 80 milliseconds. Figure 6.4b illustrates that Priority Flooding provides a real-time cloud monitoring solution.

The second experiment on the realistic topology has four flows send at a higher rate than the bandwidth limit and one flow send less than its fair share. With all five flows sending at once and an effective bandwidth limit of 6.67 Mbps on each link, the fair share of each flow is 1.34 Mbps. However, the results from the first experiment on the realistic topology show that all five flows receive more than 1.34 Mbps bandwidth. Therefore, we instead assume that each flow's fair share is actually $\frac{1}{4}$ of the bandwidth limit, or 1.67 Mbps, and have the one flow that sends at a rate less than its fair share send at 1.5 Mbps. In this experiment, the flows again start in a staggered fashion, with the flow sending less than its fair share as the first active flow.

The results of the second experiment are shown in Figure 6.5. In Figure 6.5a, we see the throughput of each flow over time. At the start, the blue flow sends at the constant rate of 1.5 Mbps. As the other flows start to send messages, the blue flow is completely unaffected and continues to experience 1.5 Mbps, while the other flows receive their fair share of the remaining unused bandwidth. Once the blue flow finishes sending messages, the other flows are dynamically allocated their fair share of the available bandwidth. Again, the fairness property and desired bandwidth guarantees are achieved. The light blue-line shows the aggregate throughput across all of the flows over time. When just the blue flow is sending, the aggregate is 1.5 Mbps. Once all five flows are active, the aggregate throughput reaches about 9.75 Mbps, due to the same protocol optimizations and latency emulation we observed in the prior experiment.



(a)



(b)

Figure 6.5: Five flows in the network with one sending less than its fair share. (a) shows throughput over time, while (b) shows factor of propagation delay over time.

Figure 6.5b shows the factor of propagation delay experienced by each flow over time. The graph shows that the maximum and mean factor of propagation delay experienced by each of the five flows in this experiment is nearly identical to those experienced in the prior experiment on the realistic topology.

6.3 Lessons Learned

By evaluating Priority Flooding on both a simple and real-world topology, we show that the protocol behavior is consistent with the desired bandwidth guarantees; all active sources receive at least $\frac{1}{n}$ of the effective path bandwidth at all times. In addition, the results show that the network resources are shared fairly among the active source nodes in the network contending for those resources. If multiple sources request more than their fair share of the available bandwidth, the bandwidth is split evenly between them. If a source requests less than its fair share, the bandwidth that is allocated to that source is unaffected by other active sources, even if those sources request significantly more than their fair share. The flow sending less than its fair share will receive all of its request and the other active sources simply share the remaining bandwidth fairly.

The evaluation shows that Priority Flooding is an effective cloud monitoring solution. The protocol achieves full network resource utilization as long as the sum of the active sources' requests are greater than or equal to the available network resources. In addition, the protocol provides real-time delivery of high priority messages, even when the network is saturated with messages. The latency experienced by the highest priority messages is not much higher than the optimal propagation delay of the underlying physical network. Lastly, Priority Flooding achieves these guarantees in the presence of compromises, making it an ideal solution for intrusion-tolerant cloud monitoring.

Chapter 7

Conclusion

The cost-effective and high-availability benefits of cloud computing are causing a wide variety of services, including critical infrastructure, to migrate to the cloud. Although the cloud offers immense cost benefits, it raises a major security concern; successful attacks against a cloud infrastructure have the effect of simultaneously disrupting all services running on that cloud. The problem of ensuring that cloud networking continues to operate correctly under all circumstances becomes important; downtime in cloud network communication equals downtime for critical services.

Accurate and timely monitoring is paramount for the correct operation of a cloud. Since cloud administrators are remote to their system, even if part of the cloud infrastructure is compromised, the monitoring system must remain operational in order for administrators to see and react to problems, prompting the need for an intrusion-tolerant messaging system.

We introduced Priority-Based Flooding with Source Fairness, an intrusion-tolerant message dissemination protocol that is designed to provide an intrusion-tolerant cloud monitoring solution. Priority Flooding provides optimal resiliency and timeliness; as long as there exists a path of correct nodes from source to destination, messages will successfully be delivered along the correct path with the shortest latency. Priority Flooding delivers messages for each source in the network in a manner that preserves source-fairness, which guarantees that each source will receive at least its fair share of network resources. If there are ample network resources, all of a source's messages will be delivered. If network resources are in contention, Priority flooding delivers a source's higher priority messages and drops its lower priority messages in order to maintain real-time delivery of the most important information to the destination.

To provide a complete solution for real-world deployment, we presented the Reliable Intrusion-Tolerant Link protocol. The protocol supplements Priority Flooding by providing reliable communication, link-level cryptographic authentication, and protection against a potentially malicious neighbor that attempts to consume a node's network resources.

We implemented both Priority-Based Flooding with Source Fairness and the Reliable Intrusion-Tolerant Link protocols in Spines, an open source overlay network framework. We evaluated our implementation on a simple network topology and a realistic network topology based on a global cloud provider's network. The evaluation showed that Priority Flooding maintains fairness among the active source nodes in the network, meets the desired bandwidth guarantee, and ensures real-time delivery of sources' higher priority messages even when the network resources are in full contention.

Bibliography

[1] R. Langner, “Stuxnet: Dissecting a cyberwarfare weapon,” *Security & Privacy, IEEE*, vol. 9, no. 3, pp. 49–51, 2011.

[2] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi, “Duqu: Analysis, detection, and lessons learned,” in *ACM European Workshop on System Security (EuroSec)*, vol. 2012, 2012.

[3] ——, “The cousins of stuxnet: Duqu, flame, and gauss,” *Future Internet*, vol. 4, no. 4, pp. 971–1003, 2012.

[4] (2013, May) Nagios. [Online]. Available: http://www.nagios.org

[5] (2013, May) Cacti - the complete RRDTool-based graphing solution. [Online]. Available: http://www.cacti.net

[6] (2013, May) Zabbix: An enterprise-class open source distributed monitoring system. [Online]. Available: http://www.zabbix.com

[7] (2013, May) Ganglia monitoring system. [Online]. Available: http://ganglia.info

[8] M. Roesch *et al.*, “Snort-lightweight intrusion detection for networks,” in *Proceedings of the 13th USENIX conference on System administration.* Seattle, Washington, 1999, pp. 229–238.

[9] C. L. Van Jacobson and S. McCanne, “tcpdump,” Lawrence Berkeley National Laboratory, 1991. [Online]. Available: http://www.tcpdump.org

[10] (2013, May) OSSEC - open source security. [Online]. Available: http://www.ossec.net

[11] G. G. Finn, “Reducing the vulnerability of dynamic computer networks,” DTIC Document, Tech. Rep., 1988.

[12] B. Kumar and J. Crowcroft, “Integrating security in inter-domain routing protocols,” *ACM SIGCOMM Computer Communication Review*, vol. 23, no. 5, pp. 36–51, 1993.

[13] Y. Rekhter and T. Li, “A border gateway protocol 4 (BGP-4),” 1995.

[14] S. L. Murphy and M. Badger, “Digital signature protection of the ospf routing protocol,” in *Network and Distributed System Security, 1996., Proceedings of the Symposium on.* IEEE, 1996, pp. 93–102.

[15] J. Moy, “Rfc 2328: OSPF version 2,” 1998.

[16] K. A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, and R. A. Olsson, "Detecting disruptive routers: A distributed network monitoring approach," *Network, IEEE*, vol. 12, no. 5, pp. 50–60, 1998.

[17] S. Cheung and K. N. Levitt, "Protecting routing infrastructures from denial of service using cooperative intrusion detection," in *Proceedings of the 1997 workshop on New security paradigms*. ACM, 1998, pp. 94–106.

[18] R. Perlman, "Network layer protocols with byzantine robustness," Ph.D. dissertation, Massachusetts Institute of Technology, 1989.

[19] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens, "An on-demand secure routing protocol resilient to byzantine failures," in *Proceedings of the 1st ACM workshop on Wireless security*, ser. WiSE '02. New York, NY, USA: ACM, 2002, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/570681.570684

[20] B. Awerbuch, R. Curtmola, D. Holmer, C. Nita-Rotaru, and H. Rubens, "ODSBR: An on-demand secure byzantine resilient routing protocol for wireless ad hoc networks," *ACM Trans. Inf. Syst. Secur.*, vol. 10, no. 4, pp. 6:1–6:35, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1284680.1341892

[21] Y. Amir, P. Bunn, and R. Ostrovsky, "Authenticated adversarial routing," in *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, ser. TCC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 163–182. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00457-5_11

[22] Y. Amir, P. Bunn, and R. Ostrovksy, "Authenticated adversarial routing: Full version," 2009. [Online]. Available: http://arxiv.org/pdf/0808.0156v3.pdf

[23] Y. Afek, E. Gafni, and A. Rosén, "The slide mechanism with applications in dynamic networks," in *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*. ACM, 1992, pp. 35–46.

[24] J. Deng, R. Han, and S. Mishra, "INSENS: Intrusion-tolerant routing for wireless sensor networks," *Computer Communications*, vol. 29, no. 2, pp. 216–230, 2006.

[25] C. Karlof and D. Wagner, "Secure routing in wireless sensor networks: Attacks and countermeasures," *Ad hoc networks*, vol. 1, no. 2, pp. 293–315, 2003.

[26] Y.-C. Hu, A. Perrig, and D. B. Johnson, "Packet leashes: a defense against wormhole attacks in wireless networks," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 3. IEEE, 2003, pp. 1976–1986.

[27] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978. [Online]. Available: http://doi.acm.org/10.1145/359340.359342

[28] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[29] H. Krawczyk, R. Canetti, and M. Bellare, "Hmac: Keyed-hashing for message authentication," 1997.

[30] W. Diffie and M. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

[31] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, "TCP congestion control with a misbehaving receiver," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, pp. 71–78, Oct. 1999. [Online]. Available: http://doi.acm.org/10.1145/505696.505704

[32] (2013, Mar.) Spines. [Online]. Available: http://www.spines.org

[33] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, pp. 1482–1493, 1991.

[34] "OpenSSL project," http://www.openssl.org, accessed: 2013-07-09.

[35] (2013, Mar.) LTN Global Communications. [Online]. Available: http://www.ltnglobal.com