A SCALABLE AGENT-BASED SYSTEM FOR NETWORK FLOW RECONSTRUCTION WITH APPLICATIONS TO DETERMINING THE STRUCTURE AND DYNAMICS OF DISTRIBUTED DENIAL OF SERVICE ATTACKS

by

OMER DEMIR

A dissertation submitted to the Graduate Faculty in Computer Science
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy, The City University of New York

2010

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

	(Bilal Khan)
Date	Chair of Examining Committee
	(Theodore Brown)
Date	Executive Officer

Bilal Khan

Ping Ji

Nancy Griffeth

Ala Al-Fuqaha

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

A SCALABLE AGENT-BASED SYSTEM FOR
NETWORK FLOW RECONSTRUCTION WITH APPLICATIONS TO
DETERMINING THE STRUCTURE AND DYNAMICS OF
DISTRIBUTED DENIAL OF SERVICE ATTACKS

by

OMER DEMIR

Advisor: Bilal Khan

In this thesis we describe a novel agent-based architecture for flow reconstruction, and demonstrate how it can be applied to obtain a description of the structure and dynamics of distributed denial of service (DDoS) attacks. We show that the system can operate in a decentralized manner, effectively providing a description of the structure and dynamics of traffic flows even with very modest levels of agent deployment. By providing structural information, the system facilitates the execution of DDoS mitigation strategies close to the actual sources of attack traffic.

Through simulations, we validate the efficacy with which the system is able to discover traffic source locations and the structure of traffic flows. Through packet-level simulations, we show favorable convergence properties for the system. We describe several schemes for selecting the precise links on which agents should be placed, and show that these placement schemes yield marked improvements in system performance and scalability. Finally, we introduce a prototype attacker localization scheme called SLANT, which combines information from a sequence of attacks on different victims, in order to further isolate traffic source locations. SLANT shows promise for using multiple attack data to determine the exact locations of the attackers, even at moderate agent deployment levels.

ACKNOWLEDGMENTS

I would like to thank the members of my committee Professors Ping Ji, Nancy Griffeth and Ala Al-Fuqaha for their guidance and feedback on my research. I would like to thank my advisor, Professor Bilal Khan for his mentorship during the research process. I would like to thank Professor Ted Brown for his support and encouragement throughout my studies as at graduate student at the CUNY Graduate Center. I would like to thank the Turkish National Police for their financial support of my doctoral studies. I would like to thank my wife, kids, parents, and friends for their continuous support and patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

We are in a world of high technology where hi–tech devices have became indispensable. Although mostly used for good, opportunities provided by hi–tech can be abused to commit crime. **Denial of service** (DoS) is an example of hi-tech crimes. DoS occurs when legitimate users are prevented from getting access to shared resources or services [25]. When a DoS attack is mounted concurrently from large numbers of distributed sources, it is called a Distributed Denial-of-service (DDoS) attack. Although DoS/DDoS attacks are just one of many challenges facing the Internet, they have been identified as the most critical concern, by Internet Service Providers (ISPs) (see for example, the Arbor Networks survey in 2008 [45]). There are several reasons for ISPs to consider DoS/DDoS as a critical concern; here we give a brief summary of them – a more detailed treatment is given by the author in [19].

First, DoS and DDoS attacks may easily be launched and require little skill. This is because there are many ready-to-use tools for conducting such attacks. According to federal prosecutors in Boston, on November 19, 2008, a teenager confessed to a three-years crime spree of controlling "Several" Botnets comprising "Tens of thousands of infected computers" used to carry out distributed denial of service (DDoS) attacks on his victims [26].

Second, the intended damage caused by DoS attacks can in scale range from specific to widespread. They can cause country-wide infrastructure problems which can disrupt all

communications on a national level. A recent example of an attack on this scale was seen on June 25th 2008, when Radio New Zealand International reported that an attack on the National Telecommunications Authority, the monopoly Internet provider of Marshall Island, caused a complete shutdown of email traffic to the country for a week [30]. Attack dynamics frequently cross national boundaries. One such case included a web site of Rapid Satellite of Miami, which is a US company in Florida which was attacked by European hackers who may have been hired by a competitor of the company [33].

Third, traffic rates seen during DoS/DDoS attacks are beginning to reach very high values. Thus, even when the intended scale is small, there can be significant collateral damage to parties not directly targeted by the attackers. According to Arbor Networks [45], a company providing secure service control solutions for global networks, the largest Denial of Service (DoS) attack recorded (as of 2008) reached traffic rates in excess of 40 Gigabits per second! This is more than 4000 times the maximum download rate of current high speed residential Internet access.

Fourth, DoS and DDoS detection is complicated by the fact that regular users may unintentionally cause network problems that result in service disruptions. For example, the Associated Press reported that the "Reply-to-all" behavior of employees of the U.S. State Department caused a denial of service, since e-mail queues were clogged; employees were subsequently warned not to use the reply-to-all option [34].

Fifth, the number of people prone to adverse side-effects of DoS attacks is increasing. On January 23, 2009, ComScore, Inc, reported that, the total number of worldwide Internet users exceeded one billion [16]. Until effective defense mechanisms are put into place, all Internet users are vulnerable to attacks.

Currently available technologies are inadequate both for protecting from attacks and for

locating the perpetrators. Faced with inadequate technological support, many victims try to resolve their problems in their own ways. For example, the technology enthusiast group Overclockers.co.uk announced on January 22nd, 2009 that it would give £10,000 ($13,830) as a reward for any information leading to the conviction of attackers who had targeted their site in prior DDoS attacks [35].

The Internet plays a significant role in our daily lives. DoS attacks are a big threat to the Internet, and are a threat which promises to get more destructive daily. Everybody is a potential victim of a DoS attack. There is a need to develop DoS defense mechanisms and doing so, respond to the technological-societal circumstance described above. In order to build effective defense systems, it is essential to catalogue DoS attack techniques and evaluate existing countermeasures.

1.1 What makes Denial of Service possible?

The way Internet is built has significant consequences for its security. In order to gain the most of the Internet, its network resources are shared among all the clients. The sharing, however, is not always fair. When some party uses resources extensively or does not obey the rules ("Protocols"), all individuals that are using that part of the network suffer from the situation. This is a classic case of the "Tragedy of the Commons" [39]. And it is this property that is intentionally abused by attackers to conduct DoS or DDoS attacks.

Below we list some core features of the Internet's design, and the implications of each on the feasibility of DoS/DDoS, and the challenge of its mitigation.

Packet switching makes IP traceback difficult. The Internet is based on packet switching. different packets may take different routes a common destinations because rout-

ing tables may change for a number of reasons such as link outage or link congestion. The possibility of multiple routes makes tracing attack packet flows more difficult. Most techniques proposed for flow tracing assume that routing tables are static during an attack.

The Internet's core components lacks the capability to provide security and authentication. The network core deals with very high volumes of packets each of which must thus be processed very quickly. This implies that core network components do as little processing as possible for each packet. This in turn implies that all the complex security-related computations must be left to edge components, and that the network core's provides few security or authentication services.

The Internet's core links have large capacity, making it easy for attackers to forward large amount of traffic and overwhelm the victims' network resources. The Internet core must support high network flow rates. This requires the core's components to be fast. The differential between fast core networks and slow edge networks is what provides attackers with opportunity to aggregate large amounts of attack traffic and forward it to the victim network, which being much lower capacity than the core network links is thereby rendered inoperative due to congestion.

The Internet's administration and management is distributed and heterogeneous, which makes deployment of DDoS defense mechanisms difficult. The Internet is composed of interconnected networks which are managed by different autonomous organizations. Defense mechanisms generally require widespread deployment in order to be effective. Decentralized Internet management makes it difficult to install defense mechanisms in a widespread manner. Also, standards bodies are slow to adopt security enhancements to protocols because of capital investments in present technologies. Vendors are reluctant to be the first to incorporate costly enhancements and features which require

en-masse adoption to be seen as worthwhile by their customers. This is a classical case of Prisoner's dilemma [51].

1.2 Common strategies for DoS power amplification

In order to make destructive attacks, the attackers have to amplify their attack power. Without this an attackers would not be successful because the attackers' own network resources are generally comparable to their victims' network resources. When an attacker tries to single-handedly overwhelm a victim, it will exhaust its own resources in the process and thus fail. Because of this, attackers need some intermediaries to amplify their attack power; we will call this the "Power Amplification Problem".

For example, in bandwidth flooding attacks, a large amount of aggregate attack traffic must be generated to overwhelm the network communication. A single attacker is unlikely to be able to create that much traffic by themselves. Even if they were able, most attackers would be hesitant to produce such voluminous traffic from a single source, since this will affect the attacker's own network and would be easily detected by security systems. Because of this, attackers employ intermediaries to help them amplify their attacks. There are two kinds of intermediaries; we call these "Unwitting Accomplices" and "Dedicated Attackers". We consider each of the types of intermediaries in turn in what follows.

1.2.0.1 Unwitting accomplices

This type of mediation uses machines which are not compromised as intermediaries. This is to say that "Regular machines" that behave according to normal protocol requirements

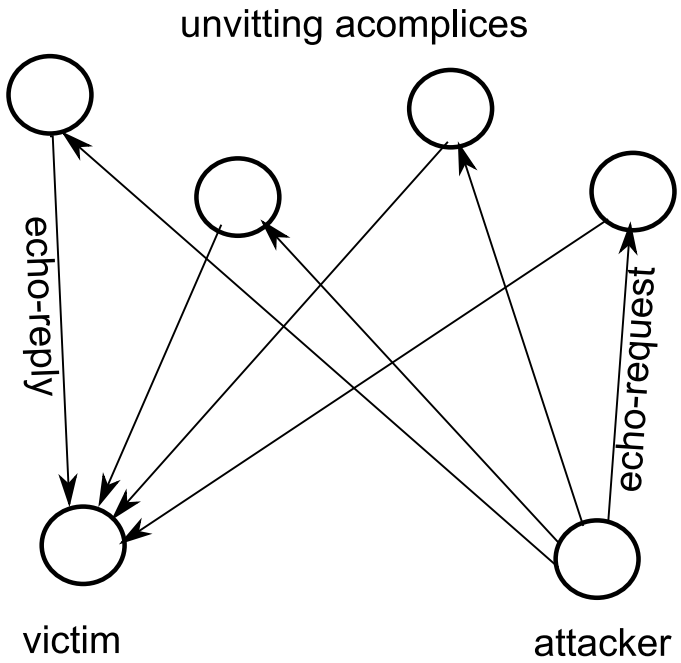


Figure 1.2.1: Sample ping attack

are somehow orchestrated by attackers so that their normal behavior creates a DoS attack on the victim. For example, as seen in figure 1.2.1 attackers send an echo-request packets with the source IP address forged ("Spoofed") to be the victim's IP address, to a large number of unwitting accomplice machines. Machines normally send echo-reply messages to the source IP address of the incoming echo-request packet, which is the IP address of the victim. If the reply traffic is high enough, the victim's network gets overwhelmed, and DoS is achieved.

1.2.0.2 Dedicated attackers

The second type of mediation uses dedicated attackers. These are compromised machines that are remotely controllable and can be made to execute commands given by central commander. There are two ways to acquire control of a computer to turn it into a dedicated attacker: directly or indirectly. In direct acquisition of control, carefully crafted packets are

sent to vulnerable systems (e.g. where the operating system has software bugs) causing the system to be compromised and henceforth remotely controllable. In indirect attacks, users are "Voluntarily" give up their machines by clicking on some link or installing malicious programs. In both cases, malicious software termed a "Bot" is installed on the compromised computer turning it into a dedicated attacker. The Bots collection of (referred to as a "Botnet") can then be used by commanders for various purposes, including to launch DDoS attacks. Once a Bot is installed, it waits for commands from its commander on the communication channel, and then executes them. Examples of commands include attack command which generates attack traffic to a particular victim, or other built in commands, which exists in most Bot software such as Stacheldraht [20]. Bots also have the capability to update themselves using software patches distributed via the command channel. Bot inter-communication is usually achieved via Internet Relay Chat (IRC) servers and channels, though it is possible for them to use other methods for communication. Nevertheless, in order to install Stacheldraht on a machine and turn it into a Bot, an attacker will typically exploit an implementation based flaw.

1.3 Classifying attack types

1.3.1 Using protocol implementation flaws

Attacks based on protocol implementation flaws exploit the weaknesses or "Bugs" in the implementation of operating systems or computer applications. For example, the "Ping of Death" attack exploits a weakness of some operating systems against over-sized ICMP packets. According to Cert Advisory CA-1996-26 [10], some systems receiving over-sized ICMP datagrams are reported to crash, freeze, or reboot, resulting in denial of service.

Exploits based on implementation flaws often lead to compromised machines which are then used as dedicated attackers or "Bots" in DoS attacks that are based on protocol design flaws.

1.3.2 Using protocol design flaws

Attacks based on protocol design flaws use the weaknesses originating from design problems of the existing Internet protocols to create a flood of network traffic. SYN Floods, Internet Control Management Protocol (ICMP) Floods, and Distributed Reflector attacks are all examples of attacks based on protocol design flaws.

The SYN Flood attack is based on protocol design flaws described in RFC4987 [21]. Briefly, in TCP communications, every computer keeps state information for all connection requests. The TCP protocol has a three-way hand shaking mechanism which uses SYN, SYN-ACK, and ACK messages to establish a connection. Whenever a SYN, SYN-ACK, or ACK packet is received, a computer updates connection state as appropriate until the final ACK, when connection is considered established. The SYN Flood attacks exploit vulnerabilities in this connection establishment protocol's finite state machine by sending a flood of SYN requests for which no ACK will ever be sent. The attacker is then able to fill the queue of unprocessed SYNs. Once the backlog exceeds its maximum allowed size, the victim's computer is unable to accept any more connections. In this state, the victim's computer can not serve even the requests of legitimate users, and the goal of DoS is achieved. The source IP of the SYN packets are usually spoofed in SYN Flood attacks.

The ICMP Flood attack is another example of attacks based on protocol design flaws. Normally, ICMP packets are used to check the network status. The Ping protocol is based

on ICMP and is intended to be used to check if a remote machine is "Alive". The querying machine sends echo-requests to the machine it wishes to probe. A machine which receives an echo-request is supposed to reply with an echo-reply packet. The "Smurf " attack is a kind of ICMP Flood attack [3]. It is conducted by sending echo-requests to a large number of machines. The source address of the echo-request is spoofed to be the victims IP. Any device receiving this message replies to the victim. This results in victim's bandwidth being exhausted by the large volume of unsolicited echo-replies.

Protocol design based attacks frequently include a "Reflection" layer leading to what Paxton [48] termed distributed reflector attacks (DRA). In these, the attackers with access to Bots gain added security by hiding themselves behind another layer of machines. The new layer is composed of reflectors, which are unwitting accomplices inserted between the Bots and the victim. Any machine that replies to regular network requests can be used as a reflector. For example, the Smurf attack described above can be used in conjunction with reflector attacks as follows: All the computers in the Botnet will be commanded to send echo-request messages to the reflectors, which will send echo-reply messages to the victim. Consequently, the victim's network is overloaded while the identities of attackers' Bots are protected. DNS Amplification Attacks are another example of Distributed Reflector Attacks. In these, legitimate DNS requests are sent to recursive DNS servers by computers in a Botnet. The source IP of DNS request packets is spoofed to be the victim's IP address. Upon receiving the requests, DNS servers dutifully reply to the victim, thereby congesting its network resources. According to Vaughn [52], an amplification factor of upto 73 can be achieved by this type of attack.

We can further classify attacks based on protocol design flaws according to their scales and the OSI layer they target (See Table 1.3.2 and Table 1.3.1). An attack can be made on scales ranging from as small as an individual/organization to as large as a region/infrastructure.

An attack can target different layers of the network stack: including the user, application, network/transport, and physical layers.

1.3.2.1 Scales of attack

DoS attacks may be directed toward companies, as was seen on January 23, 2009 when Network Solutions announced that it was having problems with all its name servers because they were under a very large-scale UDP/53 DDoS attack for more than 48 hours [55]. Attacks may also be directed toward personal pages, e.g. site of the president of Georgia was knocked offline by a distributed denial-of-service (DDoS) attack on July 21, 2008 [32]. In another example, "The Iranian opposition coordinated an ongoing cyber attack that has successfully managed to disrupt access to major pro-Ahmadinejad Iranian web sites, including the Presidents homepage" [18].

Scale of Attacks
Individual/Organization
Regional/Infrastructure

Table 1.3.1: Classifying attacks by scale

Attacks can also target regions much larger than a single computer or an organization. A number of attacks have targeted the core elements of the Internet (e.g. DNS servers). The objective of these attacks was to collapse the Internet infrastructure. For example, in 2002, all 13 root DNS servers were attacked, but the attack were unsuccessful. Another recent example which happened recently was the complete shutdown of email traffic to the Marshall Islands on June 25th, 2008 due to a DDoS attack [30].

1.3.2.2 Layers of attack

Each attack targets a specific communication layer in the OSI [69] networking stack. For example, HTTP Flood is an attack which targets the **Application Layer**. This attack is based on forcing the victim server to execute memory or CPU intensive applications (i.e. database query, document download, etc.) by a large number of clients so that the server runs out of resources. As the number of attackers increases, so does the destructive capacity of the attack.

Attack Layer
User
Application
Network/Transport
Physical

Table 1.3.2: Classifying attacks by OSI layer

SYN Flood attacks are an example of attacks which target the **Transport Layer**, because the SYN Flood attack exploits the design flaws of TCP connection establishment transport layer protocol. Similarly, ICMP Flood attacks can be classified as the attacks which target the **Network Layer** since the ICMP protocol used for ICMP Flood attacks is considered part of the network layer.

1.4 Classifying remedies for DDoS

We presented a comprehensive survey of DoS and DDoS attacks and technologies facilitating their execution. We have classified attack methods using several systems of categories. Next we seek to classify the remedies. To do this, we can begin by noting any distributed computer system can be in three states during its lifetime: "Normal operating state(NO)" when there is no attack and the system is working normally; "Attacked and unaware state

(AU)" when an attack is underway but the system has not detected that it is being attacked; finally, it is in "Attacked and aware state (AA)" if it is under attack and has detected this fact. As designers and operators of computer systems, we want to minimize the number of transitions from NO state to AU state. However, if we are in the AU state, we want to maximize the transition speed into the AA state. Similarly, we want to maximize the transition speed from AA to NO states. Figure 1.4.1 illustrates the system states, state transitions and corresponding class of remedies.

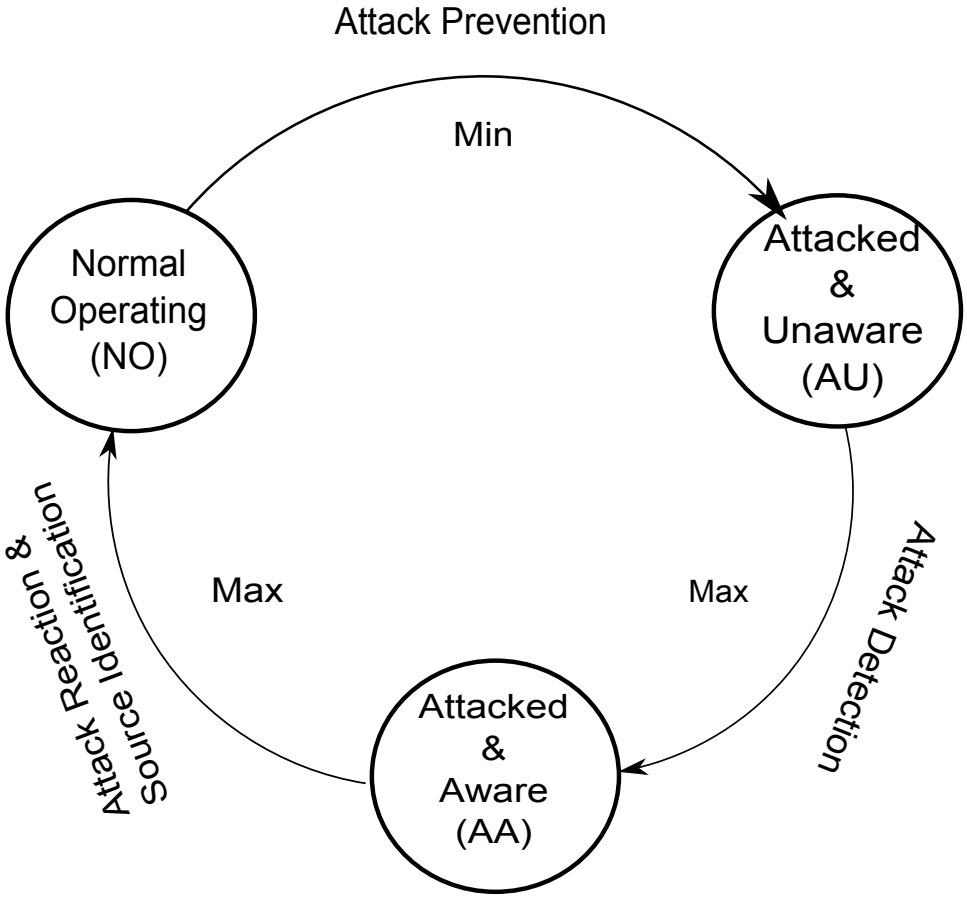


Figure 1.4.1: Attack state of a system

Among the class of remedies, **attack prevention** is located between NO and AU states. Similarly, **attack detection** is located between AU and AA states. Finally, **attack reaction** (and source identification) is located between AA and NO. This classification is used

to organize prior research efforts on DDoS, which is presented in the next chapter. We took the taxonomy development by Peng et.al [50] as a starting point but augmented it with additional considerations and dimensions.

CHAPTER 2

PRIOR WORK

DDoS mitigation has many facets, one of which is the problem of finding the true sources and structure of attacks flows . This is the problem that we will ultimately address in this research. This problem lies in the domain of attack detection, but maintains a view towards attack reaction. We begin, in what follows, by describing the landscape of prior research efforts on DDoS mitigation. We do this by considering each of the three remedy classes: (1) attack prevention, (2) attack detection, and (3) attack reaction, in turn (see section 1.4, describing their main concerns, and outlining the present approaches that have been considered in the research literature to date.

2.1 Attack prevention

The objective of **attack prevention** is to take necessary actions to minimize the possibility of the occurrence of DoS attacks. It is already mentioned that attacks are based on either design flaws or implementation flaws. The installation of security updates for operating systems and applications is the only possible insurance against implementation flaws. However, providing security against design flaws is not as simple. Researchers have proposed different types of attack prevention mechanisms for DoS/DDoS attacks based on design flaws, which we categorize here as:

- *Filtering*: In this approach, the attack traffic is filtered at routers according to their source addresses so that only packets with authorized source addresses are forwarded.
- *Overlay networks*: In this approach, the victim is hidden so that attack traffic is prevented from reaching it.
- *Charging clients for service*: In this approach clients are asked to pay for each request like doing a time consuming calculation.
- *Use of capabilities*: In this approach traffic is dropped at network routers unless they possess of valid capability tickets.

We will now briefly discuss some of the basic issues and approaches in each of these.

2.1.1 Filtering

Filtering is a process by which routers compare internal fields of incoming packets to determine whether the packets are to be processed or dropped. The most common fields considered are the source address and destination address fields in the IP header. A source IP address is considered authorized if it is reachable through the port which it arrived. There are several variations of the filtering-based approach including: ingress/egress filtering, route-based packet filtering, and source address validity enforcement protocol.

Ingress/egress filtering detects and drops spoofed packets at the network edges [23]. In **ingress filtering** approach routers forward only IP packets that have real source IP addresses, and drop packets having spoofed source IP addresses. Filtering is done by checking the source IP addresses of all incoming and outgoing packets, to determine whether or not the source address of the IP packet belongs to network subnets from which it was

received. This task requires the router to be able to check the source address of each packet against a list of candidate subnets whose outbound traffic could legitimately arrive at the router's port. It is also easy for the ISP routers (which are connecting directly to edge routers) to do ingress filtering, because all the incoming packets must have the same network address. Routers that are directly connected to host networks can easily filter traffic based on destination address, because there is generally a single IP subnet connected to such routers. this is called **egress filtering**. On the other hand, ingress or egress filtering is computationally infeasible for routers at the core of the Internet, since they are in the forwarding path of multiple and high bandwidth routes and the number of legitimate source and destination IP subnets is both large and dynamic. To be effective, ingress filtering requires wide deployment. The main shortcomings of ingress/egress filtering are its reliance on widespread adoption and its incapability to the routers at the core of the Internet and its requirement of universal adoption. In addition, ingress/egress filtering assumes that in an attack the packets' source IPs are spoofed, but this need not be the case for example when Bots are made to directly attack the victim without a distributed reflector.

In order to extend the filtering approach to the core of the Internet, a new technique called **route-based packet filtering** (RBF) has been proposed [47]. In this approach, filtering is performed at the routers that are connecting autonomous systems (AS). Each packet arriving at an incoming port is checked to determine if the claimed source IP address is reachable through that port. If it is not reachable, the packet is dropped.

The RFB technique is successful only as long as the attackers do not use carefully spoof source IP addresses in such a way that would allow the forged to pass through the filters. RBF can block DoS attacks that use random source IP addresses for spoofing, but not DDoS attacks that use real IP addresses (as is the case in reflected or Botnet driven attacks) or

attacks in which spoofed addresses are chosen with reference to the true Internet topology so as to evade triggering RBF. The RBF technique is at a disadvantage in setting where routing tables change frequently, since in such assumptions, RBF may inadvertently filter legitimate traffic and itself be the cause a denial of service.

To address this shortcoming, the **source address validity enforcement** (SAVE) protocol has been proposed [36] as an extension to RBF. SAVE requires routers to continuously broadcast messages to all their neighbors to make themselves known. Each router keeps a table which maintains current information about available routes on a per-port basis. This table provides the routers with a way to correctly and reliably authenticate the incoming packets. Although SAVE addresses the issue of route changes, it (like RBF) remains ineffective against both DoS attacks that carefully select source IP addresses, and botnet-driven DDoS attacks that uses true IPs.

2.1.2 Overlay networks

Overlay network solutions begin with the observation that hiding the IP address of a host can protect it from DDoS attacks. However, in order to be accessible to legitimate users on the Internet, a host's IP address must be known. Overlay networks serve to resolve this contradiction by providing servers with a mechanism that makes them accessible to authenticated Internet users, yet keeps them hidden from attackers. In this manner, overlay networks attempt to provide packet flow authorization for specific destinations.

2.1.2.1 Secure Overlay Services (SOS)

The Secure Overlay Services (SOS) system was designed to block DoS attacks [6]. Unlike traditional defense mechanisms that detect the attack first, and then take necessary actions, SOS does not care about detecting or reacting to attacks. It assumes DoS can overwhelm a victim machine or network as long as the location of the victim (i.e. IP address) is known. With this in mind, SOS proposes the idea of hiding the true location of the (address) victim and enforcing that only authenticated connections to the victim are allowed. The main objective of SOS is to provide a mechanism such that only authorized clients can connect to participating servers. SOS achieves this through an architecture which employs currently available Internet tools such as filters, firewalls, IP tunnels, and overlay networks.

Briefly, when a client wants to connect to a participating server, it first contacts a secure overlay access point (SOAP). The SOAP authenticates the client, after which the server hashes the target's IP, and then using a distributed hash table (Chord) [61] finds the responsible intermediate "Beacon node" and tunnels the traffic toward the beacon. The beacon tunnels the traffic to one of several "Dedicated secret servlets", which forward the traffic to the server. The return path can be either tunneled or directly connected.

Secure overlay services represents a creative solution to a complicated problem. With participating nodes and very high speed links, SOS provides adequate protection.Simulation experiments of SOS network show [6] that attackers need to target 40% of the nodes in the overlay in order to get a successful attack of probability 1/10000. SOS is thus effective for DDoS attacks. Unfortunately, however, SOS requires fundamental changes to connection establishment paradigm–users must authenticate prior to connecting. In addition, individual SOAPs are potential targets for attack.

2.1.3 Charging clients for service

If cheap production of requests creates the problem of DoS, then charging clients for each request might alleviate it. Another way of preventing attacks is by making the service requests expensive to the requester in some way. In a sense, this scheme is a form of authentication: a user is considered legitimate if they are willing to pay (something) in advance for a service. There are different approaches proposed toward charging clients for requesting services, including in terms of CPU cycles [38] or memory consumption [2]. Clients requesting service are forced to do CPU or memory intensive calculations; this is accomplished by asking clients computational questions that are easy to ask, difficult to compute the answer to (e.g the cost functions used in Hashcash [38]), and easy to validate answer to. Clients who have already consumed most of their resources for computations, won't be able to commit further attacks.

2.1.4 Capability-based prevention

Normally, any client on the Internet can ask for service from any server. Servers don't have any meaningful mechanism to drop client requests *before the requests arrive*. This is one of the causes of DDoS attacks. Capability-based prevention techniques suggest that intermediate routers can ensure that only flows authenticated by the server will arrive at the server. A flow is authenticated by augmenting its packets with special tokens are called **capabilities**, that can be used to validate the flow before it reaches the server.

Capability-based communication has two phases: connection establishment and data transmission. The connection establishment phase includes the client requesting a capability from the server. If the server decides to issue the capability to the client, the server pre-

pares the capability and sends it to the client. The data transmission is similar to regular data transmission except that the packets are augmented with capability tokens. This augmentation can be done in many ways including by using new header fields, IP options, a shim layer on top of IP, etc. When a packet with capability information arrives at a router, the router checks the capability, and forwards the packet only if the capability is valid; otherwise the packet is dropped. Capabilities can be issued permanently or temporarily. Well known clients can be issued permanent capabilities; unknown clients are issued capabilities for a limited time or in a limited amount of data.

Capability-based prevention can be effective against DDoS attacks after the authentication phase, but the authentication mechanism is open to any kind of attack. If the authentication mechanism is attacked and system can no longer provide new capabilities for newcomers, this becomes another kind of service denial attack, which is called a denial of capability attack (DoC) [7].

One extension of capability-based techniques that address DoC concerns, is the **RTS/VP Approach** [4]. This approach augments the existing Internet infrastructure with incrementally deployable request-to-send (RTS) servers joined with verification points (VPs) that are located on the data path of Internet links [5]. RTS servers are co-located at BGP routers and provide clients with the appropriate tokens needed to send packets. Verification points (VPs), are deployed near an RTS and are responsible for network bandwidth access control. VPs verify the existence of a valid token for non-RTS traffic. Domains announce their RTS servers with BGP announcements. If a client seeks to connect to a server that requires capabilities, the client must discover an RTS server, and send an RTS packet to it. RTS packets are sent via RTS servers rather than directly to the server to protect the channel used to obtain tokens against flooding. When an RTS packet reaches its destination, the destination must decide whether to allow the client to send more packets. If so, the server

calculates a capability and sends it back to the source, together with a sequence number. A capability authorizes the source to send n packets along the network path toward the destination within the next t seconds. The source labels each packet with the capability and associated sequence number. When a VP receives a packet, it checks the capability, sequence number, and flow identifiers (e.g. source and destination addresses) in its store of currently valid capabilities. The VP then forwards the packet if it has valid capability, and drops it otherwise. In order to replace expired capabilities, each VP has a mechanism to listen for capability renewals. Because RTS requests are accepted only by from clients in the RTS server's network (or from adjacent RTS servers) this protects the system from DoC attacks.

Another capability-based technique named **Stateless Internet Flow Filter** of (SIFF) technique is presented by Yaar et. al [66]. SIFF protects an end-host by selectively stopping individual flows from reaching its network. It defines two types of traffic, privileged and unprivileged (legacy) traffic. Privileged communication consists of prioritized packets subject to recipient control and is established by a capability-exchange handshake. When a source wants to create a privileged communication channel, it starts a handshake (similar to TCP). A handshake is started by sending an explorer (EXP) packet. Routers on the paths insert their information into EXP packets. When the EXP packet reaches its destination, the destination decides whether to accept the communication from the source or not. If the destination accepts, it copies the link trace information (which was created by routers en route) to the capability reply field of a new EXP packet and sends it to the source. Routers on the paths add their information again on the way back to the source. Upon receiving the reply, the source extracts the link trace information and adds it to all subsequent data packets as capability. When a router receives a data packet, it checks the capability information; if the capability contains the routers own information, that router

deletes its information from the capability and forwards the packet. Otherwise the packet is dropped.

It is not easy for an attacker to thwart SIFF by simply copying the capability of a legitimate packet because the capability contains detailed path information. Whenever an attacker sends a forged capability, any router receiving the packet will check its information and will drop the packet if it realizes that its information is not there. The attacker can be successful, however, if they reside in the same network as the legitimate user (whose hijacked capability can then be used to subvert the system's operation). Stateless filtering eliminates both the need for traditional packet filters (which is an expensive hardware resource today) and the need for any special inter-ISP relations, such as filtering agreements.

2.2 Attack detection

Mechanisms that are designed for attack detection are valuated by several criteria:

Performance criterion 1: The time from the beginning of the first attack to when it is detected. The faster attacks can be detected, the more time there is to take actions which minimize damage. Detecting attacks (and finding the attackers) can have a deterrent effect on future attacks. Detecting attacker locations (even approximately), allows us to respond in way that more effectively preserves network bandwidth from malicious users.

Performance criterion 2: The collateral damage to network bandwidth. It is imperative to detect attacks in a way that minimizes the collateral damage. The most significant challenge for detection systems which feed into attack reaction systems is false positives since these can cause unwarranted reaction and produce unintended DoS effects of their own. Because of this, defense systems often try to detect as many attacks as possible while

keeping the number of false positives to a minimum.

Attack detection is further classified into: DoS-specific versus anomaly-based detection.

2.2.1 DoS-specific detection

Mechanisms in this category depend on specific characteristics of *already known* DoS attacks to detect them. For example, SYN attacks generally do not obey TCP congestion control rules, since the attackers send as many packets as they can. This fact can be used by attack detection systems to detect SYN attacks. MULTOPS is an example of DoS-specific detection scheme [24]. This approach assumes that there is a correlation between incoming and outgoing network traffic at network edges; whenever this correlation fails to hold, it is concluded that an attack must be in progress. Another example of DoS-specific detection scheme was proposed by Blazek [53], who proposes employing a statistical analysis of data from multiple layers of the network protocol stack to detect subtle traffic changes. Blazek's approach is based on change-point detection algorithms that are computationally simple and thus can be implemented on-line. We describe a few promising DoS-specific approaches in section 2.2.1.1 and section 2.2.1.2 below.

2.2.1.1 Detecting SYN flood attacks

In "Detecting SYN Flooding Attacks", the authors propose a simple mechanism for detecting this specialized attack based on TCP [27]. The approach is both stateless and has low computational overhead. The author's approach works because it is based on the following property of the TCP: the number of SYN packets and the corresponding number of FIN & RST packets in any direction are almost equal. Normally, connections last for a certain

amount of time and end with FIN packets. Since every TCP connection requires a SYN packet to establish a connection and a FIN packet to end it, the number of SYN packets and corresponding FIN packets in one direction must be equal. This is not exactly the case in real-life situations because connections can be dropped abnormally when network problems arise. In such cases, instead of FIN packets, RST packets are used. The presence of RST packets implies that the total number of SYN packets must be approximately equal to the sum of the total number of corresponding FIN packets and RST packets. When these numbers differ "Significantly" the network is likely to be experiencing a SYN attack. The main question is what constitutes a significant difference?

In practice, network traffic quantities are different for different networks; and are even different for the same network at different times of the day. This causes a considerable difference between the number of SYN and FIN packets, and makes it difficult to determine the best settings quantifying "Significant difference". In order to make the algorithm independent of network size and time of day, a non-parametric cumulative sum (CUSUM) method is developed in [27]. The system uses a historically weighed average of the number of FIN packets to determine its decision criteria. The proposed system is capable of detecting SYN flood attacks. Several trace-driven simulations are done to check the efficacy of the detection mechanism. The results show that the detection mechanism has short detection latency and high detection accuracy.

This mechanism does not offer a new attack reaction; any defense mechanisms already available can be used. If it is deployed close to the flooding sources, the system can easily reveal attackers' location without resorting to complex IP traceback techniques. The proposed mechanism can be easily embedded into firewalls allowing most SYN flood attacks to be blocked. On the other hand, there are several serious problems associated with the system: It is very vulnerable to attackers intentionally injecting RST or FIN packets into

the attack packets and it is not effective against connectionless flooding (e.g. UDP).

2.2.1.2 Image-based approaches

Image-based anomaly detection techniques provide a way to see network traffic anomalies by human eye [31]. This research develops a simple and effective way to look at aggregate information about network traffic in order to detect anomalies. the system simultaneously detects, identifies and visualizes attacks and anomalous traffic in real-time. The main idea is to make the network traffic visual, so that the ongoing traffic flow can be monitored by humans and software. It facilitates the use of motion detection algorithms to infer the next target in the network and the use of video compression techniques to reduce data storage requirements. Scene change and pattern recognition techniques can be used to detect anomalies, while motion prediction techniques can be employed to predict attack trends.

The system has three main components. The first is a traffic parser which is responsible for image creation. The second is a data transformer which is responsible for generating traffic signals from images and analyzing the statistical properties of traffic distribution over aggregate traffic. The last is an anomaly detector. The traffic parser makes use of a two dimensional array to store packet statistics. Each row in the array can keep 255 records and there are 4 rows for the four bytes of IP address. For example, the first row of the array stores the number of packets per unit time corresponding to the first byte of observed IP addresses (i.e. a value of “4” located at the first row at an offset “192” means that the total number of packets seen to have “192” as the first byte of their IP address was 4), the second row stores the number of packets per unit time corresponding to the second byte of observed IP addresses, and similarly the third and fourth rows. Visual representation

of network traffic is achieved by arranging the normalized packet count of each of the 256 entries corresponding to each IP byte into a 16-by-16 square. Finally, four 16-by-16 squares are organized as a 32-by-32 frame. Similarly, four 256-by-256 squares are used to show the normalized values for the source and destination addresses simultaneously. Three images are created; two 32×32 images for source and destination addresses and one 256×256 image for source-destination address. Normalized packet statistics are represented by the intensity of pixels. Figure 2.2.1 shows creation of visual representation of network traffic.

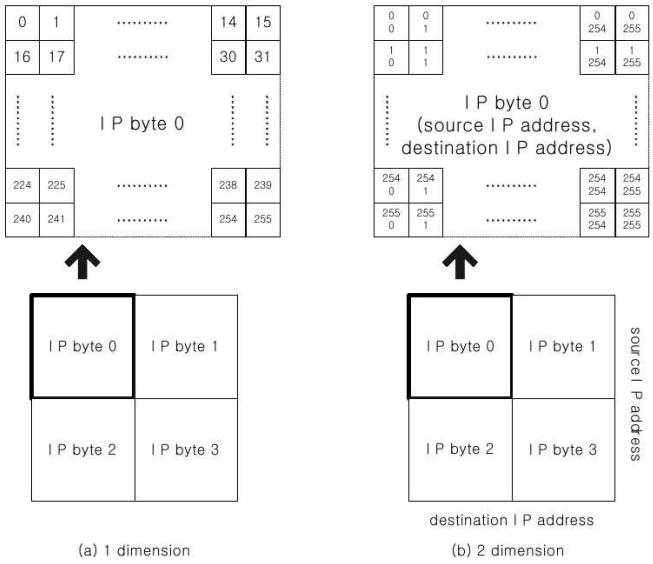


Figure 2.2.1: Image creation [31]

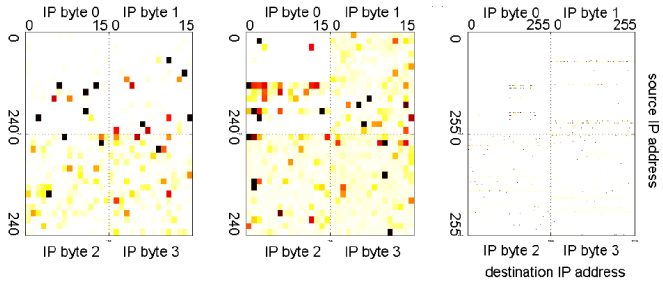


Figure 2.2.2: Image during normal network traffic [31]

In practice, images show different characteristics depending on to the network status. Figure 2.2.3 is an example screen-shot showing the network traffic during worm propagation.

Horizontal lines show that there is a traffic going toward successive IP addresses. These kinds of images are indicators of worm propagation traffic. On the other hand, in the normal network traffic there is no significant visual information detectable in the image. Figure 2.2.2 is an example screen-shot for normal network traffic. Results show that Image-based Anomaly Detection is a promising technique that might be used by network security personnel.

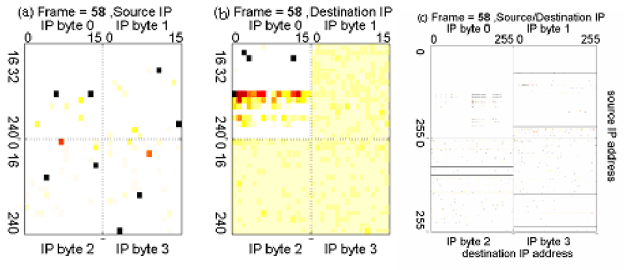


Figure 2.2.3: Image during worm propagation [31]

2.2.2 Anomaly-based detection

Unlike the DoS-specific detection scheme, an anomaly-based detection system (ADS) tries to detect all attacks, *including previously unknown ones*. An ADS begins by building a normal profile for the network using some parameters of network traffic. Then having the normal profile, it calculates the similarity between the normal profile and ongoing traffic to detect possible anomalies which may be attacks.

There are many examples of ADS based DoS detection systems. Forrest and Hofmeyr developed a network-based IDS, called Lightweight Intrusion Detection System (LISYS), based on a model of an artificial immune system (AIS) [29]. Another example of such a system is the Computer Defense Immune System (CDIS) proposed in [65]. We describe a few anomaly based approach in greater detail in section 2.2.2.1, 2.2.2.2, and 2.2.2.3.

2.2.2.1 Statistical approaches to attack detection and response

Feinstein et al. proposed a statistical approach to DDoS attack detection and response [22]. There is a mechanism to detect and respond to DDoS automatically, and uses the statistical properties of some of the packet header fields. Two values they use for this purpose are: entropy and chi-square statistics. In their system, attacks are detected by observing the variance in entropy values in network traffic streams as it shifts between the attack and non-attack states. It is claimed that there is a small variance between the entropy values when no attack is occurring, but that the entropy values of the system span a wider range when it is under attack. The entropy is calculated using different header values including source IP, destination IP, or destination port. Chi-square statistics are used to compare the distribution. In order to compute entropy, the packet field values are often binned. For example, source IPs can be binned such that the first bin contains the most frequently seen sources, and the second bin contains less frequently seen sources, and so on. The entropy is then be computed on the occupancy of the bins.

The results of Feinstein et al's research show that the entropy and chi-square tests can detect attacks whenever random source IPs are used, but when stealth IPs are used, the entropy values for non-attack and attack cases get closer to each other and the chi-square statistic shows overlaps. Many attack types are detected easily when the traffic volume is high, but very poorly when the network traffic is low. However, the detection rate of fixed-destination IP and stealth source IP attacks is very low. If the detection mechanisms are to be used as the basis for attack reaction, the system must be made capable of differentiating between the different attack types, in order to determine suitable reaction. Overall, this approach show promise for DDoS detection and could be further developed to better detect and characterize the attacks.

2.2.2.2 Biologically inspired approaches

The central feature of CDIS is a computational immune system model based on its biological counterparts. While CDIS is not an exact model of the biological immune system, it abstracts the ideas applicable to attack detection. Here we describe three main concepts used in CDIS. The first concept is "Self" and "Non-self" where self represents normal packets, and non-self represents attack packets. The second concept is "Evolutionary Computation" (EC) that tries to simulate the biological evolutionary process using a population-based model. EC relies on random variation for exploration and exploitation and is based on natural selection (the survival of the fittest). The third concept is that of "Antibody", which is a signature or pattern for detecting non-self packets. Antibodies are created by randomly selecting a protocol and some of its packet-level fields as attributes. In order to expand the pattern space "Covered" by an antibody, a random window is chosen for its non-boolean field values. After being created, CDIS tests the antibody against known self data. If any self data is seen to stand within the hyper-volume of the antibody, the antibody is discarded since it would cause false negatives. After random creation of the antibody, in order to make it cover as much non-self packet as possible without covering self packets, the hyper-volume of the antibody is changed by systematically modifying the ranges of its fields. In order to decrease the number of false negatives, a process of constimulation is defined in which the results of antibody decisions are compared against other antibodies. CDIS uses costimulation of antibodies both within and across different systems. The results acquired from CDIS research shows that the framework of biological immunology can play a useful role in organizing the architecture of intrusion-detection and DoS systems. Because it is built on dynamic notion if self and non-self, CDIS falls into the category of anomaly-detection schemes.

2.2.2.3 Signal analysis of network anomalies

In [9], the authors use signal analysis and wavelet techniques to detect anomalous network traffic. Wavelet analysis provides valuable information about the short-lived and the long-ived patterns in the network. Wavelets provide a means for describing time series data that considers both frequency and time (and are more useful than other decompositions like Fourier analysis when characterizing data with sharp spikes and discontinuities). On the other hand, it is not so easy to determine which wavelet system to use for analysis. The authors selected to use a wavelet system named Pseudo Splines which produces 3 signals out of the input signal: Low frequency (L), middle frequency (M), and high frequency (H) signals. The low frequency signal is used for detecting long-lived anomalies and the high frequency signal is used for detecting short-lived anomalies.

A measure called Deviation Score is proposed, and based on variability in H and M signals. Deviation Score begins by computing the local variability (using specified window) of H and M signals. Then, the local variability of H and M signals are combined (using a weighted sum) and normalized by the total variability to get a deviation score V. The analysis shows that when V peaks over 2.0 it indicates the presence of a short-lived anomaly with high confidence. In order to determine the effectiveness of Deviation Score, the results are compared to the Holt-Winters Forecasting [14]. In a network trace with 39 known anomalies, the Deviation Scoring found 38 anomalies while the Holt-Winters founds 37. The authors argue that this shows that the implementation works as well, if not better than the Holt-Winter Forecasting.

2.3 Attack source identification

Attack source identification is often an important secondary objective in attack reaction, since it is necessary to distinguish between malicious attacks and unintentional DoS phenomena. Identifying the source of attack packets is very difficult because the source IP field of the packets can be easily forged or "Spoofed". Moreover, current network standards do not require network devices to maintain information about paths taken by the packets, so detecting such spoofing is difficult.

We define **traceback** as "Actions taken to find the true source of a packet". Researchers have proposed different traceback techniques which are classified here as: IP Traceback by Active Interaction, Probabilistic IP Traceback, and Hash-Based IP Traceback.

2.3.1 IP traceback by active interaction

Traceback mechanisms in this category propose actively interfering with the attack traffic and tracing the attack sources based on the reactions to the interference. **Backscatter** is a kind of an active interaction technique [8] that is applied to BGP-level routers. Its main assumption is that the attack packets have spoofed source IP addresses that are selected among the reserved IP address space (e.g 192.168.x.x or 10.x.x.x). Backscatter seeks to find the point of entry of the attack packets into BGP-level Internet backbone. Towards this purpose, during an attack a backscatter server announces itself as the destination for IP addresses being used as spoof source addresses in the attack. Then, the backscatter server sends a BGP route announcement to make the destination network being attacked unreachable. Meanwhile attackers continue to send packets to the victim, but since the target is not reachable anymore, the ingress routers reply with a "Destination unreachable"

message, sending it to the source IP of the attack packets, which results in forwarding the "Destination unreachable"message to the backscatter server. Upon receiving the "Destination unreachable" message, the backscatter server can identify the entrance point of the attack packets into the BGP-level Internet backbone.

Backscatter is an innovative protocol-based defense. In case of a DDoS attack distributed along different Autonomous Systems (AS), the backscatter sever can identify several points as the input points of DDoS attack. However, using it has a huge collateral effect, because legitimate traffic to the victim will also be blocked at the BGP-level.

Link testing is another variant for IP traceback by active interaction. It requires checking the upstream routers manually in hop-by-hop fashion. This requires enabling "Input debugging", which allows an operator to filter packets from an egress port and determine which ingress port they arrived on. Input debugging is a very time consuming technique that requires collaboration among the network operators of different ISPs. It also requires that the attack continues during the link testing process. Controlled flooding is a form of link-testing traceback which does not require assistance from intermediate network operators [15]. In this approach, routers use a shared buffer for both incoming and outgoing traffic. When the buffer becomes full, the performance of the router decreases, which in turn, decreases the number of packets it forwards. This decrease can easily be detected by the downstream router. This property is exploited to recursively identify upstream routers through which the attack packets are coming.

Link testing works well with a global knowledge of network topology and long lived high volume attacks. However, because input debugging and controlled flooding are processor intensive techniques, using them will adversely effect all the other network communication going through the network links that are being tested.

Stone and Robert proposed an active interaction technique called **CenterTrack**, which uses an overlay network . Their proposal requires building a logical "CenterTrack" network of traceback capable routers for each ISP, on top of the ISPs' physical infrastructure. In the event of an attack, the victim sends a request to a centralized controller of its ISP's CenterTrack network. The controlling agent then broadcasts a message to all the edge routers telling them to mark packets going toward the particular victim. Routers then mark the messages with their information. Upon receipt of these marked messages, the ingress point of packets can be identified by the victim. Unfortunately, the system works only for a single AS, which means the victim must take additional actions to get closer to the attacker.

Router-centric Traceback schemes involve router cooperation in the traceback process. In [17], the router closest to the victim starts building the traceback path by sending an alert message to all its neighbors along with a traffic descriptor. Each router receiving the alert message starts listening on the network for traffic matching the given traffic descriptor. If any traffic which fits the traffic descriptor is seen, they report themselves to. This method is repeated iteratively until no more router reports are received. The main disadvantaged of this technique are that all the routers must be aware of the protocol, and the routers must do additional processing which may contribute to the DoS attack.

2.3.2 Probabilistic IP traceback schemes

Bellovin proposed another probabilistic IP traceback scheme named iTrace [11]. In iTrace, a router probabilistically selects a packet and sends an ICMP message to the destination of the selected packet. The ICMP message contains information about the router which sent it. A victim can build a reverse path to the real source of the attack using these messages.

2.3.2.1 Packet marking

Packet marking schemes involve making routers' add identifying information to packets that they forward in a manner that provides the victim with information about the path that the packets have taken.

Marking every packet, is not feasible because of the packet size limits. In addition, it requires processing for data addition and checksum calculation, which will unacceptably degrade the routers' performance. Alternatively, the **probabilistic packet marking** (see "Practical Network Support for IP traceback" [58]) suggests selecting the packets probabilistically before marking them. Transit router information is written into the IP packet's identification field, which is normally reserved for rebuilding fragmented packets. Whenever a router marks a packet, any existing information written by previous routers is overwritten by the new information. In order to find the full reverse path, there has to be at least one packet marked by the router closest to the attacker. The probability of the victim's seeing a packet marked by the furthest router is $p(1-p)^{(d-1)}$, where p is the probability of marking a packet and d is the distance from the marking router to the victim. The victim can build the reverse path to the attacker only if it can get enough packets from a marking router close to the victim. Alternatively, instead of adding node information, the packets can be marked with link information. In this case, the marked packets will carry two IP addresses, one for each end of the link. This requires a space for storing 72 bits (two 32-bit IP addresses and 8 bits for distance to represent the theoretical maximum number of hops allowed using IP) in the packet header.

There are several limitations of packet marking. First, the path convergence in the probabilistic packet marking with node information is too small. Second, if multiple attackers act concurrently, the probability that routers at the same distance mark packets will be

the same for different attack paths, which will make reconstructing flow structure more difficult: determining the order of the routers on the reverse path is more challenging. In the case of marking with path information, the algorithm converges faster. For example, if a router marks packets with probability 0.1, the victim requires only 75 packets to reconstruct the path to an attacker located 10 hops away. This algorithm can also efficiently discern the routes of multiple attacks, because it uses link information rather then node information. There are several problems related to using the identification field to write the path information. The first problem is that when the IPSec protocol is used, the aforementioned field will be encrypted. However, this is rarely the case because encryption requires data connection and trust setup, thus removing the need for packet tracing. The second problem is real fragmentation, which can create a big problem if the fragmentation occurs after packet marking. In this case, the identification field will be overwritten for packet marking, which will result in not being able to reassemble the fragmented packets. Finally, even if the traceback systems can trace back to the edge router of the attacker, the determination of the real MAC address of the attacker is still difficult. Worst of all, the MAC addresses can also be changed.

The effectiveness of Probabilistic Packet Marking (PPM) is studied by Park and Lee [46]. It is concluded that for single source attacks, PPM is effective at localizing the attack origin. The PPM can help the victim to effectively localize the physical source of the attack to 2 - 5 candidates. However, in a distributed DoS attack, as the number of mounted attack sources increases, the traceback scheme is rendered less effective.

2.3.2.2 Hash-based IP traceback

Hash-Based IP traceback (HBIT) is a kind of probabilistic IP traceback scheme. As described by Snoeren [57], HBIT proposes that *all the packets* are marked at network devices. HBIT is based on some assumptions: end users are resource constrained, IP traceback is an infrequent operation, the Internet routing can change, attackers are aware of being traced, IP packets can be directed to one or more destinations, duplicate packets may exist, and routers may be subverted. HBIT provides a way to trace every packet individually. For this purpose, a hash value of infrequently changing fields and the first 8 bits of the payload of each IP packet is computed. The resulting hash value is a n bit number called the *packet digest*. In order to store this set of hash values, a space efficient Bloom filter [13] is used. Specifically, k distinct packet digests are computed for each packet using different hash functions, and the set of k n-bit results are used to index into a 2^n-sized bit array which is initialized to all zeros; k bits are set to one for each packet received. Whenever a packet arrives at a router, the router computes k packet digests and sets the corresponding bits in the bit array to 1. Checking if a packet has been seen can be done in a similar way; once the k packet digests are computed for the packet, the corresponding positions of the Bloom Filter are checked. If any one of these bits is zero, that means the packet was not seen recently. If, however, all of the corresponding bits are one, then it is an indication that the packet may have been seen, though there may be cases when the bits are set by other packet digests, creating a false positive.

One of the components of the HBIT is a source path identification engine (SPIE). In SPIE architecture there are three components: data generation agents (DGA), SPIE collection and reduction agents (SCAR), and SPIE traceback managers (STM). The DGA calculates and stores the packet digest in a Bloom Filter. All of the SPIE enabled routers have DGA

agents. Each SCAR is responsible for a network, and knows its topology. When an IP traceback request arrives, which carries information about the time, a copy of the packet under investigation, and also the information about the final router, the SCARs recursively ask each router (starting from the final router) whether it has a record of the given packet. If there is positive information in the digest table, the router replies with a positive response to the requesting SCAR, and transfer its Bloom Filter to the SCAR. [1] A SCAR creates the partial IP traceback tree for its network. The STMs assemble the large scale traceback tree from the partial trees obtained by SCARs.

The main contribution of Snoeren's paper is its demonstration of the traceability of single packets with minimum memory requirements and acceptable false positive rates. If every router keeps records about the packets that it has forwarded using the HBIT technique, every packet can be traced. The SPIE can be effective if it is widely deployed.

There are several issues with hash-based IP traceback. Firstly, the traceback operation will be requested when the traffic is very heavy, which requires either an additional communication channel or prioritizing traceback related messages. Second, a copy of the attack packet, which is not normally stored anywhere, must be provided in order to compute the traceback. Third, any traceback request must be issued almost as soon as the attack starts otherwise Bloom Filters might be discarded and records get lost. Fourth, the packet fragmentation can change the first 8 bit of the packet payload resulting in different packet digests at different routers for the same attack packet making attack packets untraceable.

1. The DGA keeps the digest tables for a very short time; the tables of interest are sent to SCARs, and the others are discarded (because of memory limitations).

2.4 Attack reaction

In order to minimize the damage resulting from DoS attacks, a reaction scheme must be employed at the time of the attacks. Different types of reaction mechanisms have been proposed including "Bottleneck Resource Management", "Intermediate Network Reaction", and "Source End Reaction". We consider each in what follows.

2.4.1 Bottleneck resource management

Bottleneck resource management systems provide attack reaction at the bottleneck devices, which can be either hosts or network devices. At the host, for example, SYNcookies were introduced against SYN flooding attacks [12]. The SYNcookie technique requires modification in the TCP stack such that during the connection setup phase, instead of keeping information about the half open connections, a host produces a SYNcookie for each connection request and sends it to the client. If the client sends the cookie back to the host, the connection is established. Since no connection state is kept, it is not possible to overload the host with a SYN flood attack. A technique termed SYNkill is proposed as a way to thwart SYN flooding attacks [56]. The SYNkill technique proposes injecting RST packets into the network to kill the long lived half open connections. Using a server farm together with a load balancer is another bottleneck resource management technique used against DoS attacks. Using more than one server for a service and hiding servers behind a load balancer is an effective way to keep critical services always up. However, this solution is an expensive one which requires investment in the hardware and the network infrastructure.

2.4.1.1 Server-centric throttling

An attack reaction mechanism that can be used as a bottleneck resource management system [67] seeks to reduce attack traffic in order to protect the target servers and let them continue serving legitimate users. Upon request from the victim S, the system installs throttles at upstream routers that are located at certain distances. The system tries to keep the amount of traffic reaching the server S within a range $[L_s, U_s]$. In order to achieve this goal, the victim server continuously checks the amount of incoming traffic. Whenever incoming traffic exceeds U_s limit, the system calculates a new upper limit for the maximum amount of traffic that each router is allowed to forward to the victim. The new upper limit is sent to all of the routers which are located at distance k. The upper limit is calculated using the multiplicative decrease additive increase method, much like TCP flow control. If the current traffic is higher than the maximum allowed limit, the routers decrease the forwarded traffic by half. If the aggregate traffic amount at the victim side is less than the lower limit, the system increases the maximum allowed traffic by adding some predefined value to the previous upper limit and sends this information to routers. The routers receiving this information adjust their traffic limits accordingly. If the traffic limits are negligible, the throttle is removed.

Experiments show that the system works well when compared to the "Pushback" technique, which we will discuss in section 2.4.2.1. Two evaluation criteria were used in the experiments comparing the two techniques: the percentage of good-user traffic making it to the server and the number of routers involved in protecting the server. The topology information collected by the Internet mapping project at AT&T was used for the experiments. A total of 5,000 trace route paths were randomly selected. The results of the experiment showed that, for evenly distributed attackers, both systems initially behaved

similarly, but then level-k max-min throttling performed better in terms of the percentage of remaining network traffic of good-users. Also, in the case of unevenly distributed attacker traffic, the level k max-min technique does perform better. However, for the case of evenly distributed attackers which behave indistinguishably from regular users, neither of the systems performs well.

2.4.1.2 Defense by offense

The "Defense by offense" technique is an attack reaction mechanism [63] that is significantly different from other approaches. It is motivated by the observation that bad clients send requests at much higher rates than legitimate clients. It assumes that the bad clients exhaust all of their available bandwidth.

This system tries to allocate available resources to all of the competing clients in proportion to their service requests. Initially, since the bad clients send the majority of the service requests, they get almost all the available services from the servers. As a reaction to the attack, this system asks all the clients, no matter whether good or bad, to send as many requests as they can, and it funnels the incoming requests proportionally such that the resultant traffic includes a fair proportion of traffic from all the clients and the resultant aggregate traffic is low enough for the server to handle it. The bad clients can not send any more traffic either because of their lack of available bandwidth, or because of their fear of triggering security systems. In contrast, since good clients have been using a very low percentage of their available bandwidth, they still have much more available.

The system named "SpeakUP" exploits the aforementioned properties. Assume that there is a server having capacity c, an aggregate traffic of good requests g, an aggregate traffic

of bad requests B, and an aggregate traffic of total possible good requests G. In case of an attack, the good clients will get $\frac{g}{g+B}$ percent of the total server capacity. If $(B+g) \gg c$ and $B \gg g$ then the good clients will not get any service. If good clients send a total traffic of G, the service shares will be $\frac{G}{G+B}$ if $G \approx c$ than G can get $\frac{1}{2}$ of total capacity. Viewing the bandwidth as currency, the system tries to allocate resources to competing clients in proportion to their bandwidths. In order to provide at least as much bandwidth to legitimate users as they had been using before the attack, the server must be able to process c packets at a time such that $c \geq 2g$. Different kinds of encouragement techniques are used for promoting good clients to send as much traffic as they can. For example, "Random drops and aggressive retries (RDAR)" is a kind of encouragement technique[63]. The RDAR randomly drops the incoming requests and then asks the clients whose packets have been dropped to resend a new request. Since the bad clients are already maxed, the majority of the requests will come from good clients . In this case, the price for a service becomes the number of retries attempted for the service. Another technique is "Explicit payment channel (EPC)" type encouragement[63]. The EPC asks all the clients to open a separate channel for payment and sends byte streams. SpeakUp has a component named thinner that tracks the number of bytes that the clients send. Whenever the server is ready, the highest ranking request is sent to the server and the number of bytes for that client is reset to zero. Figure 2.4.1 shows that during an attack SpeakUp can effectively provide good users with services at rates comparable with what they had been receiving before the attack.

2.4.1.3 Adaptive defense

The paper "Adaptive Defense Against Various Network Attacks" presents an attack reaction mechanism [70] which is a dynamic defense control system. The mechanism sets

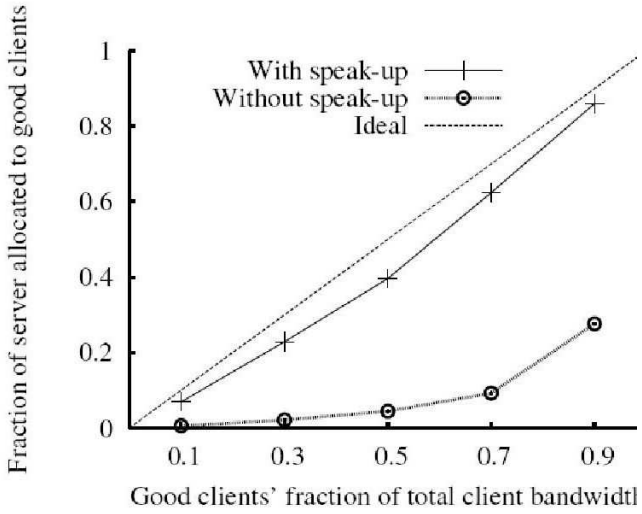


Figure 2.4.1: Behavior of SpeakUP in test environment [63]

the thresholds of underlying defense systems such that the total harm resulting from the attacks is minimized.

The threshold values of underlying defense systems can be set dynamically in such a way that **both** the total harm resulting from the attacks is minimized. The selection of threshold values directly affects the number of false positives and the number of false negatives. It is not easy to set the threshold values such that the number of false positives and false negatives are kept small. It is usually the case that if the threshold is set to keep the number of false positives low, it causes the number of false negatives to increase, and vice-versa. The adaptive defense mechanism relies on the fact that a relatively good estimate of an attack's severity can be inferred and thresholds can be set adaptively according to the severity of the attack. Two major network attacks, SYN floods and Internet worm infection are presented as concrete examples in the paper. In the SYNflood case, Extended HCF (Hop Count Filtering) is used as the underlying detection algorithm.

The following cost function is used for calculating the total cost:

$$f = \min_{\delta(k+1)} c_p[(1 - \hat{\pi}(k))]P_p(k+1) + c_n\hat{\pi}(k)P_n(k+1)$$

Where $\delta(k+1)$ represents the threshold that will be used by HCF in the next time interval, c_p stands for the cost of false positives, c_n stands for the cost of false negatives, P_p represents the probability of having false positives, P_n represents the probability of having false negatives, and $\hat{\pi}(k)$ stands for the estimated fraction of attack packets. P_p and P_n are determined by experiment on real world network data.The author the define

$$\hat{\pi}(k) = \frac{\pi\prime(k) - P - p(k)}{1 - P_n(k) - P_p(k)}$$

$\pi\prime(k)$ is the measured fraction of attack packets. $\hat{\pi}(k)$ is used to estimate the fraction of attack packets expected in the next time period. Using these formulas, the threshold values to be set are selected. Then, using the thresholds the total cost incurred as a result of the false positives and false negatives is minimized. Similarly, for the Internet worm case, a modified version of "Threshold random walk (TRW)" for the detection of the Internet worms is used. In this case, the threshold W to be used for the next time interval is calculated based on current attack severity (which is taken as, the number of host scans in the network) and selected so the total cost is minimized. This system uses a buffer aware performance function, and assumes that the servers try to accept as many requests as possible. The system activates only when the server's pending request queue is overloaded. Keeping this in mind, if the number of packets estimated to be in the buffer at the next time interval is less than the capacity of the buffer, then no filtering is imposed; otherwise filtering is activated. When calculating the expected number of packets, the expected attack rate formula is used, which is calculated using the current P_p and P_n.

The results of the experiments show that with the use of this adaptive defense mechanism, more legitimate users are serviced compared to the use of static threshold based systems. The adaptive defense mechanism can be used against DoS and DDoS attacks, but it requires good estimation of the possible attacks. Its computational overhead is small. Its adaptive parameter update includes simple estimation and optimization.

2.4.2 Intermediate network reaction

The intermediate network reaction is the name for the class of schemes implemented at intermediate network devices. The benefits of using intermediate network reaction include reduction in bandwidth wasted by attack traffic geographical isolation of attack traffic from

legitimate traffic.

2.4.2.1 Controlling aggregate flows

Aggregate-based congestion control (ACC), widely known as Pushback, is an intermediate network attack reaction mechanism [37]. Pushback acts prevents scarce upstream bandwidth from being wasted on packets that will eventually be dropped. An aggregate is a collection of packets which have same attributes (i.e. source or destination addresses, etc.) Several candidate algorithms exists by which ACC decides if the system is seriously congested, identifies the aggregate that is responsible for a significant portion of the congestion, decides how much traffic must be limited, etc. ACC is applied only when the output queue of a router experiences severe congestion, which is signaled an extended period of high loss rates at the output queue.

Pushback is used to control downstream aggregates by recursively asking upstream neighbors to rate-limit a specific aggregate. In order to change the limits set or to release the filters for aggregates that start to behave normally, the rate-limiting is updated periodically. Local ACC is triggered when the output queue experiences sustained high congestion. Using the packet drop history of the last K seconds, the ACC agent identifies the high bandwidth aggregates (based on destination IP), and computes new upper limits to which they should be restricted. A list of high-bandwidth flow destinations is extracted from the packet drop history. In order to determine the rate limit for the aggregates, the ACC agent sorts the list of aggregates based on the number of drops. It uses the total arrival rate at the output queue and the drop history to estimate the arrival rate and to calculate the excess arrival rate. Excess arriving traffic will be dropped at the rate limiter to bring the traffic rate down to the desired value. In order to report total arrival rate for that aggregate, the upstream

routers send status messages to downstream routers. These messages enable the congested router to decide if it still wants to continue Pushback or not.

The Pushback mechanism is not effective at rate-limiting against DDoS attacks that are uniformly distributed across inbound links (though a more selective extension to Pushback is proposed by Peng et al [49]). It may also overcompensate, especially during flash crowds; dropping extra traffic results in link being underutilized. It can even increases collateral traffic when legitimate and attack sources are within the same aggregate and the sources are in an edge network without Pushback. Nevertheless, local and cooperative mechanisms for aggregate-based congestion control have potential to control DDoS attacks and flash crowds.

2.4.3 Source-end reaction

The source-end reaction is the name for the class of schemes which operate at the source-end devices near attack sources. One example is DWARD [43], which collects flow statistics and periodically compares the measured statistics with normal behavior. Once a flow deviates from the normal flow model, it is classified as an attack flow and is filtered or rate-limited. Source-end reaction schemes are effective against DoS attacks because the attack traffic has similar statistical properties at both the source and the victim side. On the other hand, it is not effective against DDoS attacks because the attack traffic at the attacker source may match normal user traffic profiles.

2.5 Limitations of prior approaches

Table 2.5.1 summarizes the previous work on DDoS (which was reviewed and presented by the author in [19]) and the shortcomings of each proposed scheme. An X in a given row, column N indicates that the proposed solution violates the requirement ($N = 1, 2, \ldots, 8$ previously listed).

2.6 Looking forward

From the previous evaluation of prior approaches, we distill the following eight tenets as our requirements here:

1. **A solution that requires centralized coordination and management of DDoS defense is unacceptable.** According to CIDR Report, as of January 5th, 2010 [54], there are 33,244 unique autonomous systems (AS) in the world. Each of these is a set of connected Internet Protocol (IP) routing prefixes which are controlled by one or more network operators and presents a common, clearly defined routing policy to the Internet [28]. Given that each AS has its own network operators and its own routing policies, it is obviously impractical to coordinate a DDoS detection system across such a large number of disparate administrative entities. Furthermore, each AS may itself contain within it millions of client hosts and interconnecting routers. Recent attempts at DDoS detection have explored mechanisms which rely on centralized coordination and management, e.g. CenterTrack [59], Hash-Based IP traceback [57]. Accordingly, here **we will seek decentralized solutions for DDoS defense.**

2. **A solution should require minimal collaboration among the administrators**

	1	2	3	4	5	6	7	8
Ingress/Egress Filtering [23]			X			X		X
RBF [47]			X			X		X
SAVE [36]			X		X	X		X
SOS [6]	X		X				X	X
Charging for Service [38]								X
Capability Based [5]			X		X	X		X
RTS [5]			X		X	X		X
SIFF [66]			X		X	X		X
LISYS [29]								X
CDIS [65]								X
Statistical App Det.&Resp. [22]					X	X		X
Detecting SYN Attacks [27]			X					X
Image Based Detection [31]								X
Signal Analysis of Netw...[9]								X
Backscatter [8]	X			X	X	X	X	
Link Testing [15]		X				X		X
CenterTrack [59]	X		X	X		X	X	
iTrace [11]			X			X		
Packet Marking [58]			X	X	X	X		
Hash Based Traceback [57]			X		X	X		
SYN Kill [56]								X
Max-Min Throttles [67]	X				X	X		X
Defense by Offense [63]		X	X	X				X
Adaptive Defense [70]	X				X	X		X
Pushback [37]					X	X		X
DWARD [43]					X	X		X
An Act. Sec. Prot. Against DoS Attacks [17]	X		X		X	X		X

Table 2.5.1: Limitations of prior approaches

of different ASs. An autonomous system may adopt a policy that takes an unfavorable view of cooperation with other autonomous systems, perhaps because of concerns of competition or security. Within each AS, administrators would be expected to give priority to issues within their own domain, and would probably be unwilling to divert time and energy coordinating with other AS administrators just to determine whether an attack is taking place and what its structure is. Thus, a DDoS detection system whose efficacy is contingent on large amounts of inter-AS cooperation (at an administrative level) leaves itself unacceptably vulnerable to being undermined by the realities of overworked and sometimes reticent AS administrators. An effective system must be self-configuring across administrative domains, and the ability of the system to provide DDoS data to one domain should not require intervention and interaction with administrators of other autonomous systems. Recent attempts at DDoS detection have explored mechanisms which rely on collaboration among the network operators of different autonomous systems, e.g. link testing, Hash-Based IP traceback [57]. Accordingly, here **we will seek solutions that require minimal inter-domain collaboration during the detection phase.**

3. **A solution should yield timely, accurate quantitative actionable data concerning DDoS attacks even at modest deployment scales**. Every system exhibits an implicit trade-off curve between deployment extent and benefits provided. If the trade-off curve is such that it provides near-zero benefit until a threshold deployment level is reached, the system requires impractically large up-front investment to be beneficial. In this case, the system faces a bootstrapping problem, since it is difficult to convince administrators to join. More generally, this phenomenon is exhibited by any monotonic convex trade-off curve, since in such systems early involvement does not produce benefits commensurate with the effort required. Ideally, we would like the trade-off function to be concave monotonic, so as to motivate early adoption.

Recent attempts at DDoS detection have explored mechanisms which require large-scale deployments to be effective, e.g. Ingress/egress filtering [23], Practical Network Support for IP traceback [58], Hash-Based IP traceback [57]. Accordingly, here **we will seek solutions that are effective at producing structural and temporal data concerning DDoS attacks, even when system deployment levels are modest.**

4. **A solution that requires changing existing Internet protocols is unacceptable**. Changing existing Internet protocols requires extensive interoperability testing and lengthy approval processes within standards bodies like the IETF. In addition, there are vast existing infrastructure investments that depend on current Internet protocols, and these interests produce a systemic resistance to change. Finally, there are already large budget projects like "Clean Slate" at Stanford University which seek to [1] reinvent the Internet from scratch; the idealism that such projects take as axiomatic by far exceed our own assumptions. Recent attempts at DDoS detection have explored mechanisms which rely on changing existing Internet protocols e.g. source address validity enforcement [36], Backscatter [8], Practical Network Support for IP traceback [58]. Accordingly, here **we will seek solutions which can be built using existing Internet protocols.**

5. **A solution that requires costly modification of router internals, router firmware or router software, is unacceptable**. Modification of router internals requires cooperation of router manufacturers. While this may be possible for one or two vendors, we seek a vendor-neutral solution. Requiring widespread adoption of the functional modifications by different vendors returns us to the intractable world of standards bodies, and coordination between multiple competing vendors. Recent attempts at DDoS detection have explored mechanisms which requires modifying router

internals, e.g. source address validity enforcement [36], Backscatter [8]. Accordingly, here **we will seek solutions which require very minimal modification of router internals.**

6. **A solution that significantly increases router processing load is unacceptable**. Routers are already required to process data at astronomically high rates, and their specifications are presented in terms of maximum throughput. A solution which consumes a significant amount of link bandwidth in order to provide for DDoS detection will likely be unpopular with system administrators and router manufacturers alike, with negative consequences for adoption. System administrators would be averse to it if they saw DDoS as being a rare event that did not justify the cost of control traffic bandwidth. Recent attempts at DDoS detection have explored mechanisms which run on routers, e.g. Ingress/egress filtering [23], source address validity enforcement [36], Practical Network Support for IP traceback [58]. Accordingly, here **we will seek solutions which do not significantly increase router processing load.**

7. **A solution that requires changing existing physical network topology is unacceptable.** Router topology or interconnections is designed and implemented with regard to the functional requirements of business units. It is not feasible to change router physical/logical topology just to facilitate DDoS detection. Thus, if we can develop a technology for DDoS detection that does not require changing network topology, it will be easier to convince larger numbers of administrators to adopt our system. In turns are expecting that the higher the percentage of ASs installing our system, the higher the overall benefit will. Recent attempts at DDoS detection have explored mechanisms which rely on changing the existing network topology, e.g. Secure Overlay Services [6], Backscatter [8]. Accordingly, here **we will seek**

solutions which do not require changes to network topology.

8. **A solution that provides only a local view of the structure (spatial) and the dynamics (temporal) of a DDoS attack is unacceptable.** In order to take legal actions against the originating attackers (and the administrators of intermediate domains complicit in the attacks), structural and the temporal information is required. Most recent attempts at DDoS detection have explored mechanisms which are not capable of providing spatial and temporal description of DDoS attacks, e.g. Ingress/egress filtering [23], source address validity enforcement [36]. Accordingly, here **we will seek the solutions which are capable of providing such detailed reporting.**

2.7 Looking ahead

In this chapter, we described known approaches to mitigate DoS and DDoS. Defense mechanisms were divided into four categories: attack prevention, attack detection, attack source identification, and attack reaction–a natural division based on system state diagram of Figure 1.4.1 in the previous chapter (see pp. 12).

Attack prevention itself is divided into four subcategories: filtering, overlay networks, charging clients for service, and capability based attack prevention. Attack detection is sub-categorized as DoS specific attack detection, and anomaly based attack detection. Similarly, the attack source identification is studied under two subcategories named IP traceback by active interaction, and probabilistic IP traceback schemes. Finally, the attack reaction is divided into bottleneck resource management, intermediate network reaction, and source end reaction. Although, researchers have proposed very elegant approaches to

defend against network attacks, there is still no perfect solution.

In order to combat such attacks, it is imperative to strengthen DDoS mitigation services supported by the global network infrastructure. Moreover, global cooperation must be established for tracing international attacks. Also, ISPs must start deploying more distributed defense mechanisms at the ingress and egress points. More importantly, better DoS/DDoS defense mechanisms must be designed, developed and deployed.

We lay out the requirements of such a system in next in Chapter 3 and then design it. We define evaluation criteria for performance tuning of the system in Chapter 4, before embarking on an actual implementation in Chapter 5. The system is assessed in Chapter 6, and its performance leads to several natural extensions, which are explored in Chapters 7 and 8.

CHAPTER 3

SYSTEM DESIGN

3.1 Problem scope

In previous chapter we reviewed different DDoS remedy classes. In Section 2.1 we described attack prevention, which seeks to minimize the likelihood of successful attacks (but does not consider the problem of dealing with attacks when they eventually occur). In Section 2.2 we reviewed attack detection, whose goal is to minimize detection time, thereby facilitating rapid response (which itself is outside of the scope of detection) [1]. Finally, in Section 2.4 we covered attack reaction strategies which seek to return the system from being under attack to being once again in its normal operating state. Each class of strategies considers a segment of the life cycle of the system (see Figure 1.4.1 in the previous chapter (see pp. 12).

In this research, the problem we seek to address falls in *between attack detection and attack reaction*. We begin with the observation that attack detection, being a form of information collection, must occur at *several different scales*. On the one hand, attack detection takes place at a "small" scale whenever individual nodes perceive an anomaly, e.g. when the victim of a DDoS attack finds itself the recipient of unusually high numbers of connection requests, or when it detects a worm. On the other hand, attacks need to be detected at a

1. Section 2.3 considered the subproblem of attack source identification.

systemic scale if hope to ever obtain actionable information which can serve as the basis of effective mitigation (and litigation!) strategies. It is this question of how we can assemble microscopic local information about an attack into macroscopic global view of the attack, that we seek to address. Our presumption then, is that the problem of DoS detection has been already addressed at the local scale, and a good solution to the problem is available for us to use as a black box. Starting from this premise, we seek to develop a system to carry out the micro-to-macro (local-information to global-information) transformation.

We assume that there is a Local Detection System (LDS) which can be instrumented at any router port and can generate two types of local alerts, whose semantics translate to "There is an attack on V" and "The attack on V has ended". In practice, there are many different choices for the LDS. We will attempt to design our micro-to-macro schemes in a way that is oblivious to the implementation of the particular LDS. Wherever we need to make assumptions about the behavior of the LDS, we will state the assumptions explicitly. Two natural types of LDS which might appear include systems based on traffic volumes and systems based on payload signatures. The former could be used to trace flows involved in DDoS attacks; the latter might be used to trace the origins of particular malware. In what follows, we will assume that the LDS is volume-based. That is to say, we take for granted that each LDS generates its local alerts on the basis of traffic volume measurements.

We will assume that those responsible for DDoS attack flows (which may be Bots) hide their identities by spoofing the source IP address field in their packets. *This is the main evasion strategy that we seek to counteract.* We will assume that attackers do not use a distributed reflector array (DRA) to further evade detection. Extending the system to deal with attacker localization in the presence of DRA is briefly outlined in the Section 9.1.

Our objective is to develop an agent-based system that will:

- **Operate** scalably in a decentralized manner.
- **Require** minimal control traffic overhead.
- **Require** minimal coordination among participating autonomous systems.
- **Provide** timely, useful information concerning flow structure and dynamics of ongoing DDoS attacks, even when participation in the system is low, with the fidelity of information increasing commensurately as system adoption increases.

Within the developed system, each of the constituent agents will:

- **Self-configure** optimally and dynamically in response to attacks.
- **Use existing Internet protocols** to communicate with its peer agents.
- **Be easy to incorporate** into existing networks.

We seek to instrument an agent-based system to be able to answer these types of questions:

- What are the IP addresses of machines being victimized at any given time, through a given router?
- What time did the attacks begin?
- How long did the attacks last?
- What did the attack flow structure (tree) look like?
- What are the approximate locations of the attacking nodes?
- How did the attack tree changed during the attack?

3.2 Design assumptions

In order for an LDS to be compatible with our micro-to-macro scheme, we need to make some assumptions about the LDS. Three assumptions that we adopt include:

- The LDS generates two types of local alerts, whose semantics translate to "There is an attack on V" and "The attack on V has ended".
- If an LDS instance X raises the alert "There is an attack on V", then all downstream LDS instances Y will eventually raise the same alert (i.e. Y is on the path from X to V).
- If an LDS instance X raises the alert "The attack on V has ended", then all upstream LDS instances (which believe there is an attack on V) will eventually raise the same alert (i.e. X is on the path from Y to V).

Under suitable network assumptions, most volume-based LDS schemes can be shown to exhibit these properties. For example, the 2nd and 3rd condition require that:

1. SDA4: Network traffic is routed based only on the destination address.
2. SDA6: Routing tables of network nodes remain static during an attack.

A subset of the following assumptions are made during the experimental evaluation:

- SDA2: The false positive rate of local detection systems is 0.
- SDA3: Inter-agent messages sent over unreliable transmission protocol are not lost.
- SDA5: The false negative rate of local detection systems is 0.

We note that these assumptions will not required for our scheme to work correctly. However, in fairness, the assumptions do impact the performance of the system relative to specific metrics that we will define. Nevertheless, because we seek to quantify the *relative impact* of system parameters values on system performance, the impact of these assumptions is exhibited across different parameter setting scenarios. In short, our objective is to develop a system which achieves micro-to-macro information synthesis, and our experiments seek to determine the efficacy of our proposed solution at achieving this objective, and the sensitivity of the system to various parameters. The results obtained thus serve as a best-case analysis of our micro-to-macro information synthesis, assuming an idealized LDS.

3.3 Agent models

Because we do not wish to modify router internals, agents must be viewed as devices which reside on links. Before we design and describe how these agents inter-operate, let us define our model of what an agent is; that is to say, what it is, and is not capable of doing, and how. We propose three agent models each of which has its advantages and disadvantages.

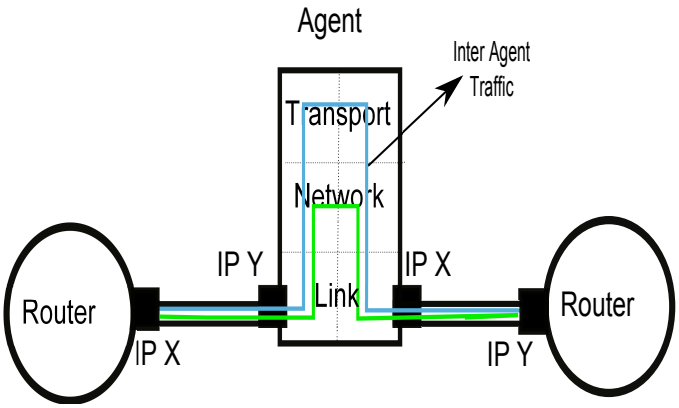


Figure 3.3.1: Inline agent model

Figure 3.3.1 shows the first, namely the “Inline Agent Model” . In this model, the agent is

implemented as a two port device installed on the link just before the routers. Note that it is completely a separate device and has no connection to the routers. The agent operates two protocol stacks, each up to the TCP transport layer. However, only the inter-agent traffic goes up to third layer, transit network traffic is only taken up to second layer. The agent does not have own IP addresses, but rather uses address X for its link to the router having a port with IP address Y, and uses address Y for its link to the router having IP address X. In this model, the agent must be able to forward the packets as fast as the router ports between which it is inserted, otherwise there will be queue buildup and packet loss, caused by the agent. Under the **inline agent model**, our proposed solution will require symmetric routing to operate properly.

SDA1: Data packet routing is symmetric. Note that this assumption is for traffic constituting the DoS attack, and does not apply to BGP routing messages.

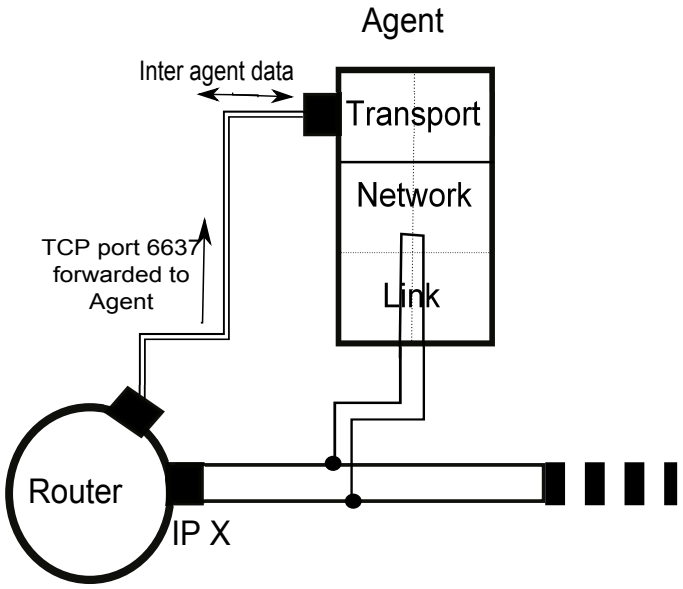


Figure 3.3.2: Network tap agent model

Figure 3.3.2 shows the "Network Tap Agent Model". In this model, the agent is implemented as a network tap which also has a dedicated direct link to a nearby router. Through

network tap, the agent can listen to the traffic in order to collect traffic statistics. Using its the connection to the router, it can communicate with the other agents. In this case, router is supposed to forward all the traffic to port number 6637 to the agent through the direct link. In this model, the agent has tapped the traffic (rather than acting as a forwarder), so it can be built using slower and cheaper hardware. The disadvantage of this model is that the router must be configured such that it forwards inter-agent TCP traffic to the agent; this adds some processing load to the routers since traffic to that interface must go up to the TCP layer.

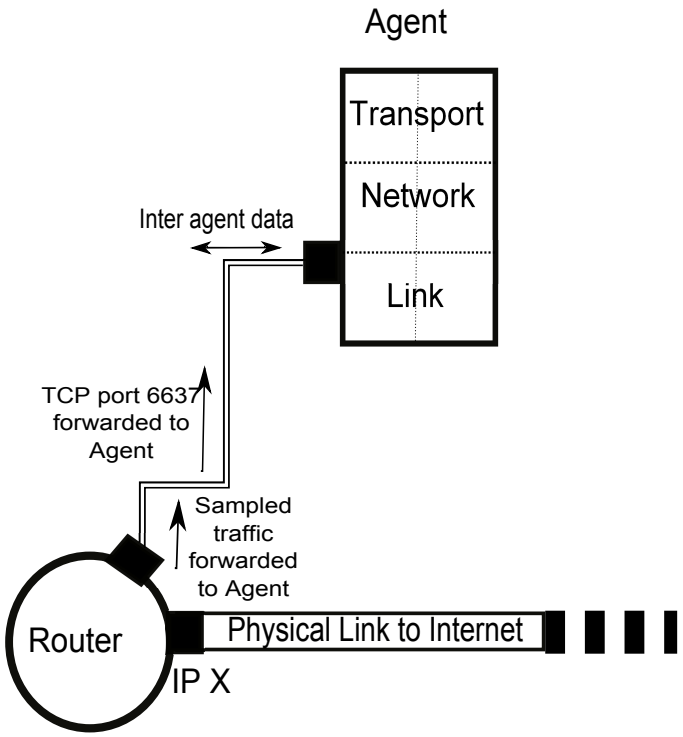


Figure 3.3.3: Router-assisted agent model

Figure 3.3.3 illustrates the third agent model. In this "Router-assisted Agent Model", the agent does not tap the link, but rather has only a direct link to the nearby router. The router is responsible for forwarding sampled network traffic to the agent so that the agent can update per-destination flow statistics. In addition, the agent uses the direct link to

communicates with the other agents. The router forwards all TCP traffic destined to port number 6637 to the agent. The disadvantage of this model is that the router must sample the traffic and send it to the agent. In addition, the router must be configured such that it forwards inter-agent TCP traffic to the agent; this adds some processing load to the routers since traffic to that interface must go up to the TCP layer.

In what follows, we adopt the Inline Agent Model, and assume network routing is symmetric[2]. The macroscopic design of the system is oblivious to which of the three agent models are selected.

3.4 System Architecture

We give an informal description of the architecture, which will be formalized in subsequent section. As described previously, we will consider the Inline Agent Model, where each agent is a two port link device which can be installed on the egress or ingress port of a router. The proposed device operates by forwarding IP traffic through itself, sampling packet headers and aggregating statistics based on destination IP address. In addition, the agent listens to all ICMP reply packets (header and payload) *regardless* of their destination address. Whenever traffic to a destination IP triggers an alarm function (e.g. the volume exceeds a system threshold) the agent creates a "Downstream Alert" message and sends it *towards the victim*. The purpose of this message is two-fold: (*i*) it informs downstream agents about a hypothesized attack on the victim, and (*ii*) it initiates the formation of logical link in an agent overlay network specifically instantiated in response to the attack. Specifically, the agent sends the Downstream Alert message as the payload of ICMP reply packet setting

2. This latter assumption is unnecessary if the system is implemented using the other agent models.

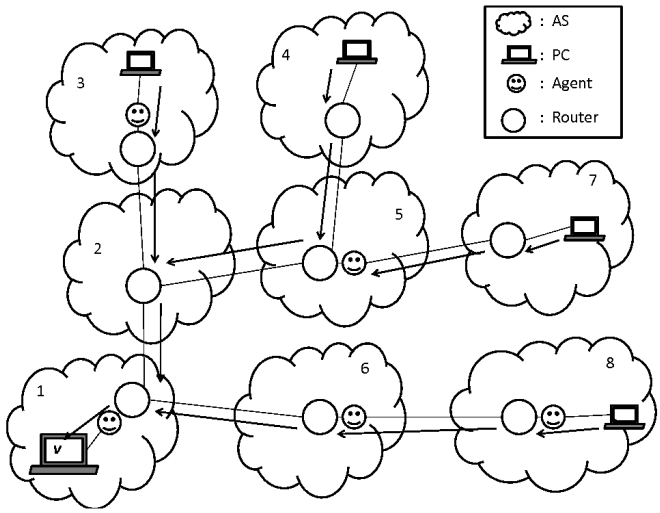


Figure 3.4.1: Sample DDoS attack with agents installed

the victim's IP address as the destination, starting with a TTL of 1, and incrementing the TTL gradually until it receives an acknowledgment from the next downstream agent in the direction of the victim. Whenever an agent sees a Downstream Alert message in an ICMP packet, it replies with an acknowledgment, revealing itself to be the next downstream agent towards the victim, and causing the upstream agent to terminate its TTL-increasing search process. The two agents can then share information concerning the attack on the victim over this newly formed link. If we view each logical link as a directed edge from upstream agents to downstream agents, the resulting logical network yields a distributed representation of a dense solution of the flow reconstruction problem. Information about the structure of this overlay network can be queried in real time by sending a broadcast message in the overlay network. Agents store their logical link history in a distributed database that can be queried to determine the structure and dynamics of attacks.

Figure 3.4.1 illustrates a DDoS attack where v is the victim, clouds numbered $1-8$ represent ASs, circles represent routers, $happyfaces$ represent agents, $straightlines$ represent $edges$ between routers, and arrows represent the attack flow. Among the ASs the ones numbered as $1, 3, 5, 6, 8$ are participating ASs which means they have installed agents somewhere in

their domains and the rest of the ASs are non participating ones. In the scenario illustrated in Figure 3.4.1, v resides in a participating network and is being flooded by attackers residing in ASs numbered as $3, 4, 7, 8$. Assuming all the agents experienced attack traffic more than its threshold, Figure 3.4.2 shows the constructed attack flow. The path to the attacker in $AS-8$ is fully reconstructed and we get the complete attack path in full detail. Although the attack traffic from $AS-3$ comes through non participating $AS-2$, with our scalable peer-to-peer agent system the path to the attacker is successfully reconstructed. In this case we lose resolution of the attack path because we do not have agents in $AS-2$ but this is less important then finding the source of the attack. The attacker from $AS-7$ can be traced back up to the $AS-5$. In addition, since agents keep records of incoming MAC addresses, we can easily go one more hop closer to $AS-7$. However, we do't have any idea if the attack is originated from $AS-7$ or not. Attacker can be in any further AS whose traffic is forwarded by $AS-7$. Similarly, we can only construct flow path up to $AS-2$ for the attacker in $AS-4$. Although we can not fully reconstruct some of the flows, the system enables traceback as far as possible with respect to the agent deployment. The reconstructed flows provide the victim with actionable information about attack structure and dynamics. For example, the attack could easily be filtered at the points closest to the attack origin, once our system has reconstructed the flow.

3.5 Agent protocols

Here we present the agent-level protocol of our agent-based architecture. We choose to represent each agent's operational protocols as a Mealy machine [40]. We describe the required transitions between states in response to events, together with any accompanying actions. Events in our agent protocol may be generated locally or may occur due to the

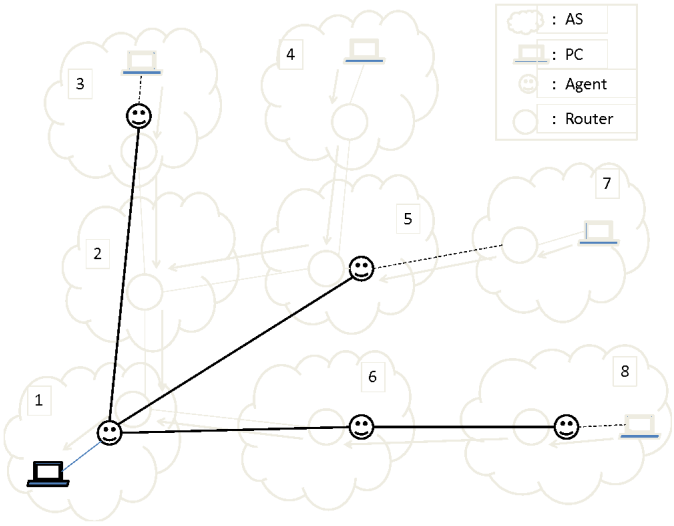


Figure 3.4.2: Sample flow reconstruction

receipt of network messages. Local events include Alert High (AH), Below High (BH), and timer expiration ($Timeout$). Message related events occur when an agent receives any inter-agent message. We have defined different messages, named: Alert Downstream(AD), No-Alert Downstream(ND), and Alert Downstream ACK (ADA). AD messages are sent using connectionless transport protocol [3] and ADA and ND messages sent using a reliable transport protocol (like TCP). In the protocol these messages will appear both as actions (when they are to be sent) and as events (when they are received).

Each agent counts the number of packets $c_v(t)$ it sees destined to each target v in a time window in the interval $(t-W, t]$. With period W it updates a sliding window estimate of the traffic to v, as $X_v(iW) = C_v(iW)+X_v((i-1)W)\cdot r$; here r is called the **statistical history coefficient** and reflects the extent to which traffic history lingers in the assessment of traffic rates. In our initial system, an $AH(v)$ event is generated locally by the agent whenever $X_v(t)$ exceeds some fixed threshold T. Similarly, a $BH(v)$ event is generated when the traffic to v goes below threshold T. Since initially all destinations are in non-alarm state,

3. This could be an ICMP-reply packet.

an AH event must occur before a BH occurs, and AH and BH events always occur in strict alteration.

Whenever an agent A gets an $AH(v)$ event it starts the process of self-organization by searching for other agents downstream towards v. The agent uses AD messages to do this, sending an AD message using connectionless transport protocol[4]. Initially A creates an AD message with $TTL = 1, Source = A, Destination = v$, sends this and sets a timer to wait for a response. If there is no response before the timer expires, a local $Timeout$ event is generated, causing A to try again with a higher TTL. This process continues until either a downstream agent B closer to v responds to A with an ADA, or the TTL value reaches 255. In the latter case, A knows that it is a **proxy to the victim**.

Whenever the agent A gets an BH event it must tell its downstream agent B that it now no longer believes there is an attack on v. The agent A does this by sending an ND message concerning v to its downstream neighbor B. However, A can only do this if the downstream agent B has revealed its identity, i.e. B has sent an ADA acknowledging the earlier AD from A. In this case, the reliable connection implementing the logical link is used between A and B is used to send the ND. If the downstream agent B has not yet sent an acknowledgment, then the ND will not be sent until the acknowledgment arrives.

Agents listen to traffic for $ICMP$ reply messages carrying AD or ND messages as their payload *regardless of the ICMP message's destination.* If an agent B sees an AD message from A to v it opens a network connection back to A (the source) using a connection oriented transport protocol (e.g. TCP)and replies with an ADA message. The ADA message from B is an acknowledgment message which tells the agent A sending AD message that there is an agent further downstream on the attack path to v. In this case, we say that A is

4. It can be sent as the payload of an $ICMP$ reply packet, or as a UDP packet, for example.

a **parent** of B. Note that in the case of DDoS, an agent will frequently have more than one parent. Thus, each agent maintains an integer variable #Parents for each victim. The variable is increased it by 1 each time an AD message to v is seen (on unreliable transport), and decremented each time an ND message concerning v is received (from one of its reliable upstream connections). An agent with no parents is a **proxy to the attacker**.

Figure 3.5.1 shows the finite state machine (FSM) of the protocol. Each circle in the FSM corresponds to a state. There are eight states, **State1, State2, State3, State4,** $State1'$**,** $State2'$**,** $State3'$**,** $State4'$, respectively.

State 1. An agent is in State 1 (with respect to vertex v) when it *neither believes that there is attack on v, nor has it received any alert messages from upstream agents* concerning an attack on v.

State 2. An agent is in State 2 (with respect to vertex v) when it believes that there is an attack on v and *is trying to find other agents on the path to v*, and has *not* received any alert messages from upstream agents, concerning this attack.

State 3. An agent is in State 3 (with respect to vertex v) when it believes that there is an attack on the v, and it has found an agent downstream in the attack with which it can communicate, but it has *not* received any alert messages from upstream agents concerning this attack.

State 4. An agent is in State 4 (with respect to vertex v) when it believes that there is an attack on v but *failed* to find any downstream agent that it can communicate with, and has *not* received any alert messages from upstream agent for this attack.

State 1'. An agent is in State 1' (with respect to vertex v) when it believes that there is no attack on v, but it has received at least one alert from an upstream agent claiming

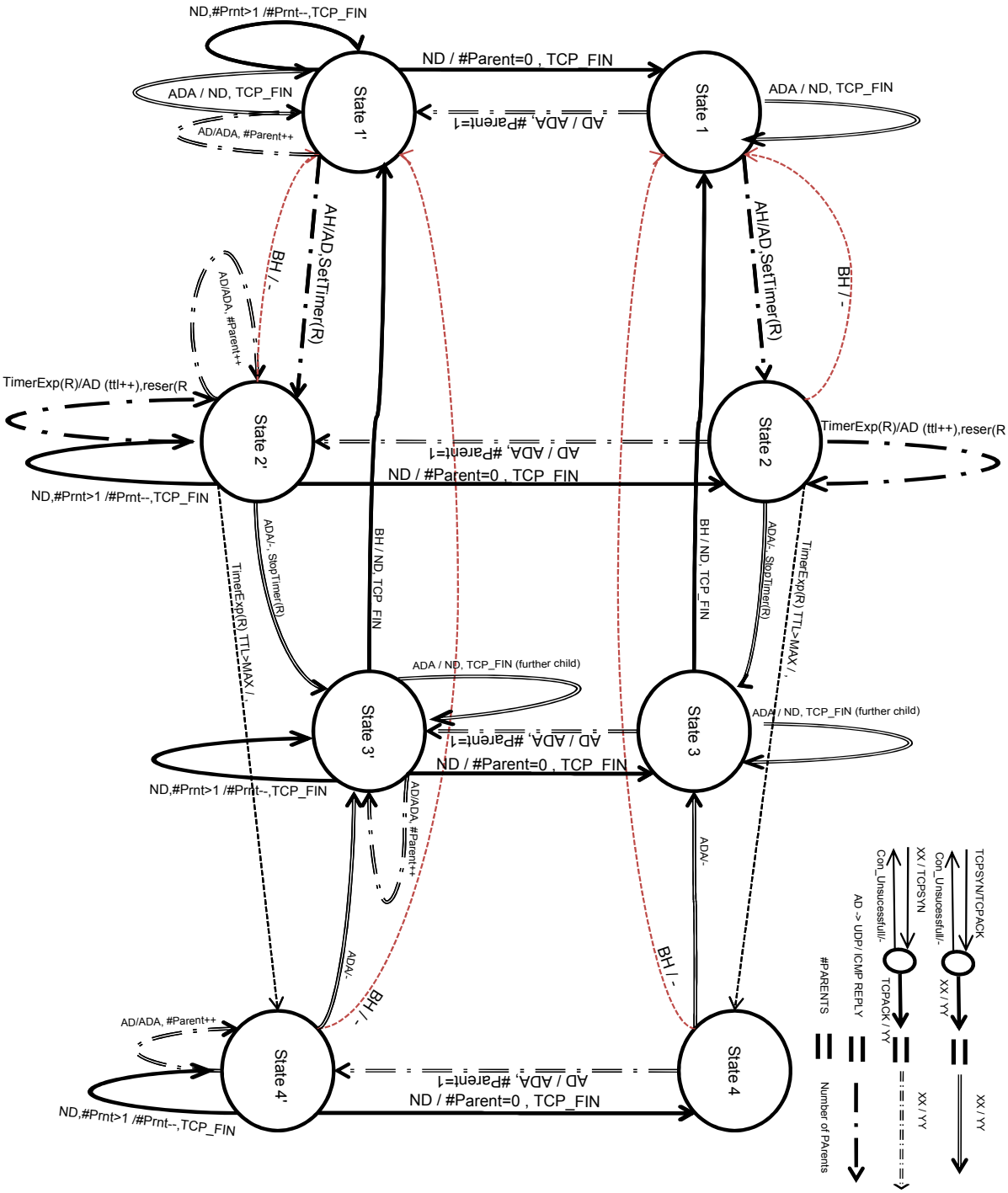


Figure 3.5.1: Mealy machine full (FSM_8) state

that there is an attack on v.

State 2'. An agent is in State 2' (with respect to vertex v) when it believes that there is an attack on v and *is trying to find other agents on the path to* v, and it has received alert messages from upstream agents concerning this attack.

State 3'. An agent is in State 3' (with respect to vertex v) when it believes that there is an attack on the v, and it has found an agent downstream in the attack with which it can communicate, and it has received alert messages from upstream agents concerning this attack.

State 4'. An agent is in State 4' (with respect to vertex v) when it believes that there is an attack on v but *failed* to find any downstream agent that it can communicate with, and has received alert messages from upstream agents concerning this attack.

Each directed link corresponds to a state change. The label over the link specifies the incoming messages and corresponding actions taken. Links depicted by double lines correspond to communication over TCP, where both the incoming and the outgoing messages are sent reliably. Similarly, links represented by dashed double lines corresponds to a communication where the incoming message was received over an unreliable channel, but the outgoing message is sent over TCP. In order to make it easier to read the messages and the actions taken, the following two tables show the initial and final states of agents corresponding to incoming messages.

Table 3.5.1 (resp. 3.5.2) shows the next state of (resp. the actions taken by) the system with respect to its current state and the incoming messages.

Initial State	**Incoming Messages**				
	AH	**BH**	**AD**	**ADA**	**timeout**
$State1$	State2	NA	$\widehat{State1}$	State1	NA
$State2$	NA	State1	$\widehat{State2}$	State3	State4
$State3$	NA	State1	$\widehat{State3}$	State3	NA
$State4$	NA	State1	$\widehat{State4}$	State3	NA
$\widehat{State1}$	$\widehat{State2}$	NA	$\widehat{State1}$	$\widehat{State1}$	NA
$\widehat{State2}$	NA	$\widehat{State1}$	$\widehat{State2}$	$\widehat{State3}$	$\widehat{State4}$
$\widehat{State3}$	NA	$\widehat{State1}$	$\widehat{State3}$	$\widehat{State3}$	NA
$\widehat{State4}$	NA	$\widehat{State1}$	$\widehat{State4}$	$\widehat{State3}$	NA

Table 3.5.1: Final states corresponding to initial state vs incoming messages

State	**Incoming Messages**				
	AH	**BH**	**AD**	**ADA**	**t.out**
$State1$	AD $SetTimer(R)$	NA	ADA $\#Parent = 1$	ND TCPFIN	NA
$State2$	NA		ADA $\#Parent = 1$	$StopTimer(R)$	$TTL \leq MaxTTL$ AD(TTL++) $SetTimer(R)$
$State3$	NA		ADA, $\#Parent = 1$	ND TCPFIN	NA
$State4$	NA		ADA, $\#Parent = 1$		NA
$\widehat{State1}$	AD $SetTimer(R)$	NA	ADA $\#Parent++$	ND TCPFIN	NA
$\widehat{State2}$	NA		ADA $\#Parent++$	$StopTimer(R)$	$TTL \leq MaxTTL$ AD(TTL++) $SetTimer(R)$
$\widehat{State3}$	NA		ADA, $\#Parent++$	ND TCPFIN	NA
$\widehat{State4}$	NA		ADA, $\#Parent++$		NA

Table 3.5.2: Actions taken corresponding to initial state vs incoming messages

3.6 Persistent state at agents

Every agent has an associated recording station, located somewhere in the wide area. A recording station is a database server which logs changes to the agent's states. A single recording station can be shared by multiple agents, or each agent can have its own recording station, or there can be a more non-uniform distribution of recording stations to agents. While the locations of the agents are secret, the locations of the recording stations are public. A special "Recording Station (RS)" record is added to DNS records by participating autonomous systems in order to provide its easy lookup of the recording stations corresponding to the agents closest to the AS. Whenever an agent FSM undergoes a state transition (including self loops), the agent requests its recording station to append a row into its **STATELOG** table. In this table, each row reflects a traversal of an arc in the FSM. Each row of the STATELOG table contains six fields; listed below and shown in Table 3.6.1.

- **Agent**, the ID of the agent writing this record.
- **Time**, the time that this record was added to the table.
- **VictimIP**, the address of the victim for which the agent logged a state change.
- **State**, the new FSM state at the agent for the victim flow.
- **Parent List**, is the new list of parents of this agent.
- **Parent RS List** is the recording stations of all agents in Parent List.

3.7 Query handling

Each recording station runs a HTTP server which responds to queries using a database engine that stores a STATELOG table described earlier. In order to expose the attack dynamics and structures, the recording stations (RS) respond to several different queries.

- Victim Search Query (VSQ): sent to a recording station (RS) to obtain a list of addresses who were being victimized at specific time.
- Attack Duration Query (ADQ): sent to a recording station (RS) to obtain the start time and duration of a recent attack on a specified victim (with the notion of recent being specified in the query).
- Tree Building Query (TBQ): sent to a recording station (RS) to obtain the attack flow tree for a recent attack on a specified victim (with the notion of recent being specified in the query).

All three queries must be directed to an RS. Thus, the querying entity must first obtain the identity of the *RS* server is identified for the given domain. This could be done, for example, by querying the *DNS* server, if DNS records were suitably augmented with RS information.

Agent	**Time**	**VictimIP**	**State**	**Parent List**	**Parent RS List**

Table 3.6.1: The agent's STATELOG table, as stored at its recording station

3.7.1 Searching for victims

The Victim Search Query (VSQ) is sent to a recording station (RS) to obtain a list of addresses who were being victimized during a specific time interval. A VSQ has the form:

$$VSQ(QID, t).$$

Each VSQ has a unique query identifier QID; the bundle of query results are tagged by this same QID to support asynchronous query submission. When a recording station receives a VSQ, it examines its STATELOG table and retrieves all rows for which $Time \leqslant t$, grouped by *Agent* and then grouped by *Victim*. For each nonempty group the RS checks the latest record and returns *Agent*, *State*, *Time*, and *Victim* if the *State* is not *State*1. If, however, no latest row has the $Time \leqslant t$ and $State\ ! = State1$ then the RS responds with sentinel values for *Agent*, *State*, $Victim = Null$, *Time*, and the QID.

Upon receiving the response to VSQ, the querying entity can consider the *Victim* fields of all the returned records show the addresses known to have been victimized at specific time by agents reporting to the queried RS.

3.7.2 Determining attack intervals

The *Attack Duration Query (ADQ)* is sent to a recording station (RS) to obtain the start time and duration of a recent attack on a specified victim (with the notion of recent being specified in the query). An ADQ has the form:

$$ADQ(QID, t, v).$$

Each ADQ has a unique query identifier QID; the bundle of query results are tagged by this same QID to support asynchronous query submission.

When a RS receives a ADQ, it examines its STATELOG table and retrieves all rows for which the *Victim IP* $= v$ and *Time* $\leqslant tmax$ and *State* is *State*2 or *State*2$'$, grouped by *Agent*.

For each nonempty group the RS finds the latest record, adding *Agent*, *State*, *Time as TimeAttackStart*, and *Victim* to a temporary table (TEMPTABLESTART). For each record in TEMPTABLESTART let $v =$ *Victim IP*, $t =$ *Time as TimeAttackStart* and $a =$ *Agent* and retrieve from STATELOG table all the records for which *Victim IP* $= v$ and *Agent* $= a$ and *State* is *State*1 or *State*1$'$. Of we find the latest record, adding *Agent*, *State*, *Time as TimeAttackStop*, and *Victim* to a temporary table (TEMPTABLESTOP). We then select the agents in TEMPTABLESTART for which there is no later entry in TEMPTABLESTOP. These are the agents which are witnessing an attack at time t. Sorting this list in descending order and let s be the time associated with the first row. Return s. The duration of the attack is then easily computed by the querying entity, as $t - s$.

Upon receiving the response to ADQ, the querying entity can consider the *TimeAttackStart* field to determine the attack start time and the computed values of *TimeAttackStop* $-$ *TimeAttackStart* to be the attack duration. If $TimeAttackStop < TimeAttackStart$, the attack is still in progress.

By making a sequence of Attack Duration Queries we can compute the start time for a sequence of cascaded attacks. The following pseudocode achieves this if we choose ϵ to be a small positive number.

`Cascaded-ADQ`(QID, time t, victim v)

- $s_0 = ADQ(QID_0, t_0, v)$, $i = 0$
- while $(s_{i+1}! = s_i)$
  - $s_{i+1} = ADQ(QID_{i+1}, s_i + \epsilon, v)$
- return s_i

3.7.3 Reconstructing flow trees

The *Tree Building Query (TBQ)* is sent to a recording station (RS) to obtain the attack flow tree for a recent attack on a specified victim (with the notion of recent being specified in the query). . A TBQ has the form:

$$TBQ(QID, t, v).$$

Each TBQ has a unique query identifier QID. The results of the TBQ are tagged by the QID to support asynchronous query submission. When a recording station receives a TBQ, it examines its STATELOG table and retrieves all rows for which the *Victim IP* $= v$ and *Time* $\leqslant t$, grouped by *Agent*. For each nonempty group the RS checks the latest record and returns *Agent*, *State*, *Time*, *Parent List* and *Parent RS List* if the *State* ! $=$ *State*1, together with the QID. If, however, there are no rows for which the *Victim IP* $= v$ and *Time* $\leqslant t$, then the RS responds with *Parent List* $= ()$, *Parent RS List* $= ()$, sentinel values for *Agent*, *State*, *Time*, and the QID.

Upon receiving a TBQ, an RS executes the following procedure:

`Process-TBQ`(QID, time t, victim v)

- Determine the *RS* servers RS_v for the v's domain via the *RS* record from DNS.

- Set $n = 0$.
- Let $G = (V, E)$ be the empty graph.
- Set *RSs-to-query* $= \{RS_v\}$.
- For each recording station z in the set *RSs-to-query*:
  - Increment n.
  - Remove z from the set *RSs-to-query*.
  - Send a $TBQ(n, t, v)$ to z. Wait for the response which will contain some number of rows, which we denote as r_n. Each row will have *Agent*, *State*, *Time*, *Parent List* and *Parent RS List* fields; the last of these two fields are lists which necessarily have the same length; we denote this length as k_j (for $j = 1, \ldots, r_n$).
  - For each i from 1 to r_n, consider the ith response row.
    - Let x_i be the *Agent* field specified in the ith response row.
    - Make a node corresponding to x_i, if it doesn't already exist; add it to V.
    - For each p from 1 to k_i:
      - Take y_{ip} to be the pth entry in the *Parent List* in the ith response row.
      - Make a node corresponding to y_{ip}, if one does not already exist, and add it to V.
      - Make a link from y_{ip} to the node corresponding to x_i, and add it to E.
      - Consider the recording station z_{ip} that appears as the pth entry in the *Parent RS List* specified in the ith response row.
      - Add z_{ip} to *RSs-to-query*.
- return (QID, G).

3.7.4 Tracing back to attackers

In order to find the closest location of the attackers we use the tree building algorithm to create the attack flow tree and then determine the leaf agents within this tree. Leaf Agents represent the closest agents to attack sources.

3.7.5 Time-lapse animations of flow trees

The purpose of animate tree-change algorithm is to show how the attack tree evolved during an attack which is occurring at time t. Let r be the frame refresh interval (measured in real time) which specifies the time between two consecutive frames. We use an ADQ to determine the attack start time and then use a sequence of TBQ requests to build the attack sequence of trees. These are used to build successive frames in an animation of the flow tree. The following procedure achieves this:

`Animate`(QID, time t, victim v)

- Use ADQ to get t_s
- Initialize $i = t_s$.
- As long as $i < t$:
  - Use $TBQ(QID, i, v)$ to obtain the flow tree T_i at time i. Add the tree to the animation sequence.
  - Increment i by r.

This algorithm provides us with $\frac{t-t_s}{r}$ separate attack trees starting form time t_s to t. The sequence of attack trees shows us how has the attack tree changed over time. Using these,

we can easily create an animation depicting attack dynamics over time using standard graph layout tools like Dot or Jiggle.

CHAPTER 4

EVALUATION METHODOLOGY

Before we can hope to quantify the performance of the proposed system, it is necessary to formally describe the problem that the agents are attempting to solve. Only then can we define the structure of a solution, and performance measures that quantify the solution quality with respect to the problem instance.

4.1 Flow reconstruction problem

To give a formal definition of the **flow reconstruction problem** (FRP), we need some preliminary definitions concerning network flows and the properties of routing tables. Given a graph $G = (V, E)$ representing a network on nodes V (and $E \subset V \times V$ represents a set of undirected edges between nodes), we introduce:

Definition 4.1.1. *A **routing table** is a function $R : V \times V \to V$ satisfying the condition that for all $u, d, g \in V$, if $R(u, d) = g$, then*

- *if u and d are distinct and in the same connected component of G, then $(u, g) \in E$,*
- *if u and d are distinct and in different components of G, then $g = u$,*
- *if u and d are not distinct, then $u = g = d$.*

Given

- Graph $G = (V, E)$ representing a network on nodes V (and $E \subset V \times V$ represents a set of undirected edges between nodes),
- Routing table $R : V \times V \to V$,
- Vertices $\{d, v\} \in V$,

Definition 4.1.2. *Given vertices $v, d \in V$ the n^{th}* ***flowstep from*** *d* ***to*** *v,* ***denoted*** *$f(d, v, n)$ is defined inductively for every non-negative integer n, as follows:*

- $f(d, v, 0) = d$,
- $f(d, v, n + 1) = R(f(d, v, n), v)$.

Definition 4.1.3. *The sequence of flowsteps $F(d, v) = (f(d, v, i); i = 0, 1, ...)$ is called the* ***flow*** *from d towards v with respect to routing table R.*

The flow $F(d, v)$ is said to be ***eventually x*** *if there exists some k such that $\forall i > k, f(d, v, i) = x$. The flow F(d,v) is said to be* ***constantly x*** *if $\forall i > 0, f(d, v, i) = x$.*

Definition 4.1.4. *A routing table R is said to be* ***consistent*** *if for all d, v in V the flow $F(d, v)$ is either eventually v or constantly d.*

Definition 4.1.5. *A routing table R is said to be* ***symmetric*** *if for all u, v in V the flow $F(u, v)$ is the reverse sequence of the flow $F(v, u)$.*

In the rest of this thesis we assume a symmetric consistent routing table. The assumption of symmetric routing can be dropped if we consider one of the alternate agent models (not the Inline Model, see pp. 59).

We are now ready to give a formal definition of the flow reconstruction problem.

Definition 4.1.6. *An instance of the* **flow reconstruction problem** *(FRP) is a tuple* (G, R, A, D, v) *where*

- $G = (V, E)$ *is a network on nodes* V *(and* $E \subset V \times V$ *represents a set of undirected edges between nodes),*
- $R : V \times V \to V$ *is a routing table,*
- $D \subseteq V$ *is a set of attackers,*
- $v \in V$ *is the victim,*
- $A \subseteq E$ *is the set of agents*

Definition 4.1.7. *A* **valid solution** *to a flow reconstruction problem is a logical network* $L = (S, E_S)$ *on a subset of agents* $S \subset A$ *and* $E_S \subset S \times S$*, where:*

$$
\begin{array}{lrcl}
1) & e \in S & \Rightarrow & \exists d \in D \\
 & & & \exists i \in \mathbb{N} \\
 & & & e = (f(d, v, i), f(d, v, i+1)) \wedge e \in A \\
 & & & \\
2) & (u, v) \in E_S & \Rightarrow & \exists d \in D \\
 & & & \exists i \in \mathbb{N}\ s.t\ u = (f(d, u, i), f(d, u, i+1)) \wedge u \in A \\
 & & & \exists j \in \mathbb{N}\ s.t\ v = (f(d, v, j), f(d, v, j+1)) \wedge v \in A \\
 & & & i < j \\
 & & & \forall k\ i < k < j\ (f(d, v, k), f(d, v, k+1)) \notin A
\end{array}
$$

Stating the above conditions informally: (1) Every agent in the solution set S lies on the flow from some attacker to the victim; (2) If two agents are connected by a logical link in L then they appear successively in the flow from some attacker to the victim and there exists no agent in between them.

Definition 4.1.8. *A valid solution* $L = (S, E_S)$ *is said to be* ***maximal valid*** *if*

$$
\begin{array}{lrcl}
1) & e \in S & \Leftrightarrow & \exists d \in D \\
 & & & \exists i \in \mathbb{N} \\
 & & & e = (f(d,v,i), f(d,v,i+1)) \wedge e \in A \\
 & & & \\
2) & (u,v) \in E_S & \Leftrightarrow & \exists d \in D \\
 & & & \exists i \in \mathbb{N}\ s.t\ u = (f(d,u,i), f(d,u,i+1)) \wedge u \in A \\
 & & & \exists j \in \mathbb{N}\ s.t\ v = (f(d,v,j), f(d,v,j+1)) \wedge v \in A \\
 & & & i < j \\
 & & & \forall k\ i < k < j\ (f(d,v,k), f(d,v,k+1)) \notin A
\end{array}
$$

Stating the above conditions informally: (1) Every agent in the solution set S lies on the flow from some attacker to the victim; and every agent that stands on the flow from some attacker to the victim is in the solution set S; (2) If two agents are connected by a logical link in L then they appear successively in the flow from some attacker to the victim and there exists no agent in between them; and if two agents appear successively in the flow from some attacker to the victim and there exists no agent in between them, then they are connected by a logical link in L. We shall see that there is always a *unique* maximal valid solution–and hence we may refer to this as the *maximum* valid solution.

Theorem 1. *Every instance of FRP* (G, R, A, D, v) *has a unique maximal valid solution.*

Proof. We begin by showing that there is a unique maximum set of agents S_{max}. Let $L_{max} = (S_{max}, E_{Smax})$ be a maximal solution, where $S_{max} \subset A$ and $E_{Smax} \subset S \times S$. Assume there exists another maximal set of agents T_{max} where $T_{max} > S_{max}$. Let h be the agent such that $h \in T_{max}$ but $h \notin S_{max}$. If $h \in T_{max}$ and T_{max} is a maximal set of agents, then by definition if $h \in T_{max}$ then $\exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d,v,i), f(d,v,i+1)) = h$. Since we assumed S_{max} is another maximum valid agent set, the definition of a maximum valid solution requires that if $\exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d,v,i), f(d,v,i+1)) = h$ then $h \in S_{max}$. But we assumed $h \notin S_{max}$, a contradiction; thus, there can not be two maximal valid agent sets. Consequently, There is a unique S_{max} for any FRP instance.

Next, we show there is a unique maximum set of edges E_S. Let $L_{max} = (S_{max}, E_{Smax})$ be maximal solution where $S_{max} \subset A$ is the set whose existence is verified in Part 1, and $E_{Smax} \subset S \times S$. Assume there exists another maximum set of edges H_{Smax} where $H_{Smax} > E_{Smax}$. Let (x, y) be the edge such that $(x, y) \in H_{Smax}$ but $(x, y) \notin E_{Smax}$. If $(x, y) \in H_{Smax}$ and H_{Smax} is a maximum set of edges, then by definition valid solution requires that if $(x, y) \in H_{Smax}$ then $\exists d \in D$ and $\exists i \in \mathbb{N}$ $s.t$ $x = (f(d, v, i), f(d, v, i{+}1)) \wedge x \in A$ and $\exists j \in \mathbb{N}$ $s.t$ $y = (f(d, v, j), f(d, v, j + 1)) \wedge y \in A$ and $i < j$ and $\forall k$ $i < k < j$ $(f(d, v, k), f(d, v, k + 1)) \notin A$. Since we assumed E_{Smax} is another maximal valid edge set, we know $\exists d \in D$ and $\exists i \in \mathbb{N}$ $s.t$ $x = (f(d, u, i), f(d, u, i{+}1)) \wedge x \in A$ and $\exists j \in \mathbb{N}$ $s.t$ $y = (f(d, v, j), f(d, v, j + 1)) \wedge y \in A$ and $i < j$ and $\forall k$ $i < k < j$ $(f(d, v, k), f(d, v, k + 1)) \notin A$. But then $(x, y) \in E_{Smax}$. But we assumed $(x, y) \notin E_{Smax}$, so there is a contradiction; there can not be two maximal valid edge sets on the vertex set S_{max}. Together, the above arguments show there is a unique maximal (and hence, maximum) valid solution $L_{max}(S_{max}, E_{Smax})$ for each instance of FRP. □

4.2 System correctness

We need to restate the system design assumptions that were stated earlier in Section 3.2. These will be necessary to prove various properties of the agent architecture we described in Section 3.4.

System Design Assumptions

- Data packet routing is symmetric (SDA1). Note that this does not include the routing of BGP protocol related messages. It is only needed when considering the inline agent model.

- Additional assumptions for a valid solution are generated by the agent system:
  - The false positive rate of local detection systems is zero (SDA2).
  - Inter-agent messages sent over an unreliable transmission protocol are not lost (SDA3).
  - Network traffic is routed based on the destination address (SDA4)
- Additional assumptions for showing that a maximum valid solution is generated by the agent system:
  - The false negative rate of local detection systems is zero (SDA5).
  - There is a static routing table (SDA6).

We will show:

- The agent system produces a valid solution, under assumptions SDA1-SDA4.
- The agent system produces a maximum valid solution, under assumptions SDA1-SDA5.

Theorem 2. *The agent system produces a valid solution, under assumptions SDA1-SDA4.*

Proof. • The first requirement of a valid solution, that we must demonstrate, is that if $e \in S$, then

$$\exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d, v, i), f(d, v, i+1)) = e.$$

The proof is by contradiction. Assume $e \in S$ but $\nexists d \in D$ such that

$$\exists i \in \mathbb{N} \ \ |(f(d, v, i), f(d, v, i+1)) = e$$

. In other words agent e is in solution set S but it is not on a flow from any attacker d to victim v. If $e \in S$ then $SDA2$ requires that there was an attack flow $F(X, v)$ going through e where $X \in D$. In other words if agent e is in solution set then according to the assumption $SDA2$ there must be a real attack flow that is seen by the agent e which made it appear in solution set s. Recall that $F(X, v) = (f(X, v, j); j = 0, 1, 2, ...)$. Since e has seen $F(X, v)$ then

$$\exists j \; s.t. \; f(X, v, j), f(X, v, j + 1) \in F(X, v) \wedge (f(X, v, j), f(X, v, j + 1)) = e.$$

In other words, since e has seen the attack flow F it must be an edge at some distance j to the attacker d. Since $F(X, v)$ is an attack flow and $X \in D$; assuming $j = i$ the $(f(d, v, i), f(d, v, i + 1)) = (f(X, v, j), f(X, v, j + 1))$ then $X = d \Rightarrow d \in D$. Consequently, $\exists d \in D$. But we assumed $\nexists d \in D$. This is a contradiction. Then

$$\forall e \in S \Rightarrow \exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d, v, i), f(d, v, i + 1)) = e.$$

- The second requirement of a valid solution, that we must demonstrate, is that if $(u, v) \in E_S$, then $\exists d \in D$ and $\exists i \in \mathbb{N} \; s.t \; u = (f(d, u, i), f(d, u, i + 1)) \wedge u \in A$ and $\exists j \in \mathbb{N} \; s.t \; v = (f(d, v, j), f(d, v, j + 1)) \wedge v \in A$ and $i < j$ and $\forall k \; i < k < j \; (f(d, v, k), f(d, v, k + 1)) \notin A$.

  Since $E_S \subset S \times S$ if $(u, v) \in E_S$ then $u, v, \in S$. We have already shown that if $u, v \in S$ then

  $$\exists i \in \mathbb{N} \; s.t \; u = (f(d, u, i), f(d, u, i + 1)) \wedge u \in A$$

  $$\exists j \in \mathbb{N} \; s.t \; v = (f(d, v, j), f(d, v, j + 1)) \wedge v \in A$$

  Suppose towards contradiction that the desired assertion is false. Assume that

$(u, v) \in E_S$ and $\exists k$ such that $i < k < j$ and $(f(d, v, k), f(d, v, k+1)) \in A$. Let $g = (f(d, v, k), f(d, v, k+1))$. Our protocol, by design, requires that if $u \in S$ then u sends an AD message towards v. $SDA4$ requires that if $(f(d, v, i), f(d, v, i+1) = u \wedge (f(d, v, k), f(d, v, k+1)) = g$ then $(f(u, v, (k-i)), f(u, v, (k-i+1)) = g$. Which means if traffic which is sent by upstream node d to target v via agent u visits downstream agent k then any packet sent by agent u towards v visits downstream agent g. $SDA4$ and $SDA3$ requires that g receives AD. Our protocol design requires that whenever an agent receives an AD message it reply with an ADA. $SDA1$ and $SDA3$ requires that u receives ADA. Our protocol design requires that when u receives ADA it checks if it already has a child and keep the child that is closer and create a logical link with the closer child and remove the further child and logical link to it. Since $k < j$ then g is closer than v consequently $(u, v) \notin E_S$, but $(u, v) \in E_S$. This is a contradiction. Consequently, $(u, v) \in E_S \Rightarrow \exists d \in D$ and $\exists i \in \mathbb{N}$ $s.t$ $u = (f(d, u, i), f(d, u, i+1)) \wedge u \in A$ and $\exists j \in \mathbb{N}$ $s.t$ $v = (f(d, v, j), f(d, v, j+1)) \wedge v \in A$ and $i < j$ and $\forall k$ $i < k < j$ $(f(d, v, k), f(d, v, k+1)) \notin A$.

□

Theorem 3. *The agent system produces the maximum valid solution, under assumptions SDA1-SDA5.*

Proof. • We have already proved the forward implication of the first requirement. Here we will prove $\forall e \in S \Leftarrow \exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d, v, i), f(d, v, i+1)) = e$.

Suppose towards contradiction that the assertion is false. Assume $\exists d \in D$ such that $\exists i \in \mathbb{N}$ $(f(d, v, i), f(d, v, i+1)) = e$ but $e \notin S$. In other words, there is an attacker d which attacks on v and there is an agent on the path from attacker to the victim but the agent is not in the solution set. Since $\exists d \in D \wedge \exists i \in \mathbb{N}$ $(f(d, v, i), f(d, v, i+1)) = e$

then e is on the path of an attack packet from d. $SDA6$ requires that all attack packets sent from d to v follow the same route then e is on the path of all the attack packets from d to v consequently e sees all the attack traffic. If $\exists d \in D \wedge \exists i \in \mathbb{N} \;\; (f(d,v,i), f(d,v,i+1)) = e$ then $SDA5$ requires that e detect the attack. If e detects the attack then $e \in S$. But we assumed that $e \notin S$. This is a contradiction. Consequently, $e \in S \Leftarrow \exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d,v,i), f(d,v,i+1)) = e$.

- We have already proved the forward implication of the second requirement. Here we will prove only the second direction of the statement; in other words if $\exists d \in D$ and $\exists i \in \mathbb{N}$ $s.t$ $u = (f(d,u,i), f(d,u,i+1)) \wedge u \in A$ and $\exists j \in \mathbb{N}$ $s.t$ $v = (f(d,v,j), f(d,v,j+1)) \wedge v \in A$ and $i < j$ and $\forall k \; i < k < j \; (f(d,v,k), f(d,v,k+1)) \notin A$ then $(u,v) \in E_S$.

  Suppose towards contradiction that the assertion is false. Assume that $\exists d \in D$ and $\exists i \in \mathbb{N}$ $s.t$ $u = (f(d,u,i), f(d,u,i+1)) \wedge u \in A$ and $\exists j \in \mathbb{N}$ $s.t$ $v = (f(d,v,j), f(d,v,j+1)) \wedge v \in A$ and $i < j$ and $\forall k \; i < k < j \; (f(d,v,k), f(d,v,k+1)) \notin A$ then $(u,v) \notin E_S$. In other words there is an attacker d attacking on victim v whose attack traffic goes through agent u and agent v, however, although there is no other agent between u and v, there is no logical link in E_S between u and v.

  We have already shown that if $\exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d,v,i), f(d,v,i+1)) = e$ then $e \in S$ and if $e \in S$ then $\exists d \in D \wedge \exists i \in \mathbb{N} \mid (f(d,v,i), f(d,v,i+1)) = e$. In other words if an agent e is on a path from attacker d to victim v then e is in solution set S and if an agent e is in solution set S then it must be on a attack path from attacker d to victim v. For current requirements this results in $u, v \in S$ but $k \notin S$. Since $u = (f(d,u,i), f(d,u,i+1)) \wedge v = (f(d,v,j), f(d,v,j+1)) \wedge i < j$ our protocol design requires that u send AD towards v. Since there is no agent k in between u and v and in S and $SDA3$ the v receives AD message. Since v receives AD message

our protocol design requires v sends ADA to u. If v sends ADA to u then $SDA1$ and $SDA3$ requires that u receives ADA. Our protocol design requires that when u receives ADA it checks if it already has a child and keep the child that is closer and create a logical link with the closer child and remove the further child and logical link to it. Since there is no agent k in between u and v, a logical link is created between u and v which means $(u, v) \in E_S$. But we assumed $(u, v) \notin E_S$. This is a contradiction. Consequently, if $\exists d \in D$ and $\exists i \in \mathbb{N}$ $s.t$ $u = (f(d, u, i), f(d, u, i + 1)) \wedge u \in A$ and $\exists j \in \mathbb{N}$ $s.t$ $v = (f(d, v, j), f(d, v, j + 1)) \wedge v \in A$ and $i < j$ and $\forall k$ $i < k < j$ $(f(d, v, k), f(d, v, k + 1)) \notin A$ then $(u, v) \in E_S$.

□

4.3 Parameters

Inputs to the attack flow reconstruction problem are (G, R, D, v, A) where $G = (V, E)$ is a network on nodes V (and $E \subset V \times V$ represents a set of undirected edges between nodes), $R : V \times V \to V$ is a routing table, $D \subseteq V$ is a set of attackers, $v \in V$ is the victim, $A \subseteq E$ is the set of agents.

Thus, there are many parameters which could influence system performance; some of these will be considered in the course of evaluating the proposed system. Others are slated for further study at a later date.

- **Graph parameters:** We will consider Waxman models [64] as our random sampling model for generating graphs. It has its own set of specialized parameters which influence graph connectivity. Common to all generative models is the parameter

of graph size, which we will certainly consider in order to assess scalability of the proposed system. We will also consider subgraphs from Cooperative Association for Internet Data Analysis [68], which provides tools and analysis promoting the engineering and maintenance of a robust, scalable global Internet infrastructure.

- **Routing table parameters:** Is the routing table consistent? If not, how inconsistent is it? Does the routing table encode the shortest path metric? If not, what is the stretch factor? Is the routing table static? If not, how frequently does it change? Does it pass through inconsistent states when it changes? Initially, we will restrict our attention to static consistent routing tables which completely agree with the shortest path (min-hop) metric. If time allows, we will consider the effects of loosening each of these assumptions (in some quantifiable way).

- **Attacker parameters:** What is the absolute number of attackers? What is the density (relative number) of attackers? What is the distribution of the attackers in the network? What is the volume of each attacker node's traffic?

- **Victim parameters:** Where is the victim? How are the successive victims are selected?

- **Agent parameters:** What is the absolute number of agents? What is the density (relative number) of agents? What is the distribution of the agents in the network? What is the threshold (or other alarm parameter) for each agent node?

4.4 Performance measures

System performance measures fall into two classes: **Steady-state** measures and **transient** measures. The former concern the behavior of the system under steady stimuli, typically

when sufficient time has passed for the system response to stabilize, e.g. How accurately do the reconstructed flows reported by the system agree with the true flows involved in the attack once the attack is fully underway? The latter class of performance measures involve aspects of the system that are intrinsically related to the pre-stabilization behavior, e.g. How long did it take the system to reconstruct a significant fraction of the attack's structure after the start of the attack? Steady state measures can sometimes be estimated by an analysis of the steady state inputs (if the protocol is well behaved); determining transient state measures almost always require protocol-level simulations.

4.4.1 Steady-state measures

A performance measure is a function that evaluates the quality of a solution (S, E_S) with respect to a problem instance (G, R, A, D, v). We need to establish some preliminary notations using which we can define our performance measures. In order to establish the necessary notation to define our performance measures, we begin by defining a function $d_{G,R,v} : V \times V \rightarrow \mathbb{N} \cup \{\infty\}$. Intuitively, $d_{G,R,v}(x, y)$ equals the number of hops that a packet takes to reach y when it is sent by x to v in graph G according to routing table R. Formally

$$d_{G,R,v}(x,y) = \begin{cases} 0 \text{ if } x = y \\ k \text{ if } (x \neq y) \wedge (y \in F(x,y)) \wedge (k = \min_j\{(j+1)|f(x,v,j) = y\} \\ \infty \text{ if } (x \neq y) \wedge (y \notin F(x,v)) \end{cases}$$

Note that $d_{G,R,v}$ is not generally symmetric or transitive, and hence does not define a metric on V. Now given any $Y \subset V$ and $x \in V$, we define the distance from x to Y

$$d_{G,R,v}(x, Y) = \min_{y \in Y}\{d_{G,R,v}(x, y)\}$$

The set of **undiscovered** attackers $U \subset D$ is defined as

$$\{u \in U \quad | \quad d_{G,R,v}(u, S) = \infty\},$$

and the discovered attackers are then simply taken as the complement $D \setminus U$. With all this notation in hand, we are now ready to define two performance measures by which to assess the quality of a solution with respect to a specific problem instance.

The **undiscovered attacker rate**

$$M1((G, R, A, D, v), (S, E_S)) = \frac{|U|}{|D|}.$$

When M1 is zero, every flow from every attacker is intercepted and hence detected by some agent. When M1 is one, every flow from every attacker is goes undetected by the agent system. Clearly, lower values of M1 are preferred.

We define the **mean distance to discovered attackers**

Definition 4.4.1.

$$M2 = MD(G, R, D, v, A, S, E_S) = \frac{\sum_{d \in D \setminus U} d_{G,R,v}(d, S)}{|D \setminus U|}$$

and use this to define the **mean normalized distance to discovered attackers**

Definition 4.4.2.

$$M3(G,R,A,D,v),(S,E_S) = \frac{1}{|D \diagdown U|} \sum_{d \in D \diagdown U} \frac{d_{G,R,v}(d,S)}{d_{G,R,v}(d,v)}$$

When M3 is close to zero, every attacking flow that has been intercepted by an agent has been intercepted close to the attacker. In this case, traceback succeeds in getting close to the attack sources. When M3 is close to one, every attacking flow that is intercepted by an agent has been intercepted close to the *victim*. In this case, traceback fails to reach the attack source. Clearly, lower values of M3 are preferred.

Of these measures, M1 and M3 lie in the range [0,1] while M2 lies in the range $[0, |V|-1]$. Note that M2 and M3 all concern how well we can "Capture" the discovered attackers, while M1 actually measures the extent to which the attackers remain undiscovered. Low values are desirable for all measures, however, M1 is more significant than the others, since a high value of M1 (regardless of the values of M2, M3 and M4) signifies a system for which a large fraction of the attacking nodes remain undiscovered.

Example 4. *In Figure 3.4.2 (reporoduced again here as 4.4.1), nodes* $A1$*,* $A2$*,* $A3$*, and* $A4$ *are attacking to victim* V*. In this case* $A1$ *is discovered by the agent in cloud 3,* $A2$ *is discovered by the agent in cloud 1,* $A3$ *is discovered by the agent in cloud 5, and* $A4$ *is discovered by the agent in cloud 8. This makes* $|U| = 0$*. Since* $|D| = 4$*, in this example* $M1 = \frac{0}{4} = 0$*, which means all the attackers are discovered.*

Since $d_{G,R,v}(A1,S) = 0$*,* $d_{G,R,v}(A1,v) = 3$*,* $d_{G,R,v}(A2,S) = 4$*,* $d_{G,R,v}(A2,v) = 4$*,* $d_{G,R,v}(A3,S) = 1$*,* $d_{G,R,v}(A3,v) = 4$*,* $d_{G,R,v}(A4,S) = 0$ *and* $d_{G,R,v}(A4,v) = 3$*, it follows that* $M3$ *for this example is* $M3 = \frac{1}{3}(\frac{0}{3} + \frac{4}{4} + \frac{1}{4} + \frac{0}{3}) = \frac{5}{12}$*.*

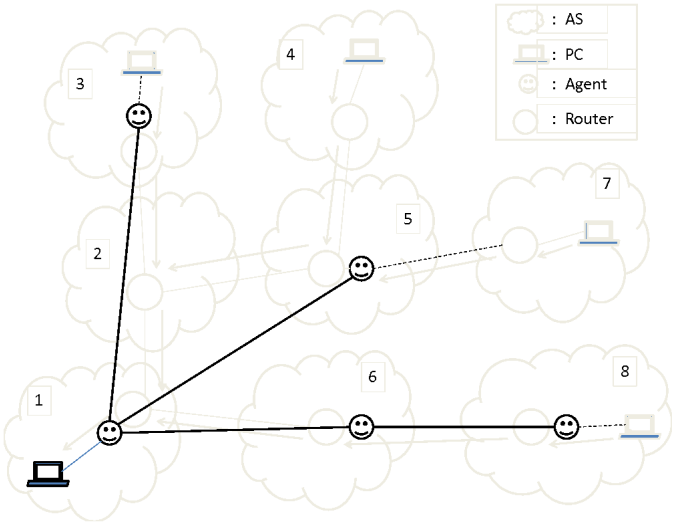


Figure 4.4.1: Sample flow reconstruction

4.4.2 Mean values of performance measures

Unfortunately, in practice, we do not know where the attackers $D \subset V$ lie in G, nor do we know which victim they will choose to target. DDoS attacks are frequently orchestrated by botnets, and thus involve arbitrary sets of attacking nodes located all over the Internet which collude to attack the chosen victim. Because we do not know the locations of the attackers or victims, the M1 and M3 performance measures defined in the previous section cannot be directly computed.

Starting from our definition of M1, we seek a derived performance measure that depends only on G, R, and A. This measure, denoted $E[M1]$, is the expected fraction of attackers which will be discovered when a random set of nodes collude to attack a random victim. Note that the **expected fraction of undiscovered attackers** $E[M1]$ on the triple (G, R, A), can be computed as:

$$\sum_{D\subseteq V,|D|=1,v\in V} \frac{M1((G,R,A,D,v),\ s(G,R,A,D,v))}{|V|^2} \tag{4.1}$$

Similarly, instead of M3, we use the expected mean normalized distance to attacking nodes. This measure quantifies the answer to the following question: If a random collection of attacking nodes were to attack a random victim, then on the flows which were intercepted by our agents, what is the expected value of the normalized distance from our agents to the attackers? This quantity, **expected normalized distance to discovered attackers**, denoted $E[M3]$, may be computed for a triple (G, R, A) as follows:

$$\sum_{D\subseteq V,|D|=1,v\in V} \frac{M3((G,R,A,D,v),s(G,R,A,D,v))}{|V|^2\cdot(1-E[M1])} \tag{4.2}$$

4.4.3 Transient measures

In what follows, assume $G = (V, E)$ is a network where V is the set of network nodes, and E is the set of undirected edges, i.e. Bidirectional network links between nodes.

Definition 4.4.3. ***Traffic history*** *is defined as a function $c : V \times R^+ \to R^+$ For a vertex (v) at time t, the traffic seen is $c(v, t)$. We will assume that c can be modeled as a continuous function.*

Definition 4.4.4. *We say that* ***traffic is eventually stable at v***

$$\textit{iff } \exists t_0 \in R^+ \textit{ such that } \forall t > t_0, c(v,t) = c(v,t_0)$$

If traffic is eventually stable at v, then we define:

Definition 4.4.5. *We denote the* ***traffic convergence time at v*** *as* $t_c(v)$*, and define it to be:*

$$t_c(v) = min\{t_0 | \forall t > t_0, c(v,t) = c(v,t_0)\}$$

The ***stable traffic at v*** *is denoted* $\overline{c}(v)$ *and is defined as*

$$\overline{c}(v) = c(v, t_c(v))$$

Consider a set of agents $A \subset V$ distributed in the network $G = (V, E)$ each of which is running a protocol (FSM) having a set of states S. We codify the state of the agents at any point in time as follows:

Definition 4.4.6. *The* ***state history*** *of a set of nodes A, each running an instance of a protocol with states S defines as a function:*

$$\sigma = V \times R^+ \rightarrow S.$$

For a vertex (v) at time t, the state of the system is $\sigma(v,t)$. Recall that our protocol generates local events AH and BH based on whether locally measured traffic volumes exceed or go below a threshold value. Thus, the function σ depends on c. When we seek to draw attention to this dependence, we will write $\sigma(v, t_c)$. Accordingly, we say that

Definition 4.4.7. *The* ***state is eventually stable at*** v

$$if \ \ \exists t_0 \in R^+ \ such \ that \ \forall t > t_0 \ \ , \ \sigma(v,t) = \sigma(v,t_o)$$

If state is eventually stable at v, then we define:

Definition 4.4.8. *S3: We denote the state convergence time at* v *as* $t_\sigma(v)$*, and define it to be:*

$$t_\sigma(v) = min\{t_0 | \forall t > t_0, \sigma(v,t) = \sigma(v,t_0)\}$$

The **stable state at v** is denoted $\overline{\sigma}(v)$ and is defined as

$$\overline{\sigma}(v) = \sigma(v, t_\sigma(v)).$$

Definition 4.4.9. *A protocol is said to be* ***stable*** *if*

$$\{\forall v \in A, \ \ traffic \ is \ eventually \ stable \ at \ v\} \ \ \Rightarrow \{\forall v \in A, \ \ state \ is \ eventually \ stable \ at \ v\}$$

Definition 4.4.10. *Given an instance of the flow reconstruction problem* (G, R, D, v, A)*, where each of the agents is running a stable protocol* P*, we define the* **convergence time** *of* P *to be the maximum divergence between the traffic convergence time at the victim, and state convergence time. Formally*

$$conv(P, G, R, D, v, A) = \max_{a \in A} \{t_\sigma(a) - t_c(v)\}$$

In practice, evaluating this measure requires us to instrument a protocol-level simulation of a DDoS attack by the nodes of D on the victim v. We do this by assuming that the

attackers D do not ramp up their attack, but rather all coincidentally and instantaneously, start sending SYN packets at a constant attack rate f that exceeds the agent thresholds. In the course of the simulation, we record the traffic history and state history in order to determine when the traffic at the victim stabilizes, and how long after that the state history stabilizes for all the agents. Such measures allow us to evaluate the latency between traffic convergence time and state convergence times in the agent network.

4.5 Formal analysis of the protocols

Sometimes it is possible to formally deduce certain properties of a distributed system through a careful analysis of its constituent protocols. In particular, the following two properties are of interest in distributed systems of agents running FSMs which implement flow reconstruction over a traffic-volume based LDS:

- A protocol is said to be **stable** if whenever traffic volumes converge to a steady state, the states of the agent FSMs also stabilize. This concept was already defined in Section 4.4.3.
- A protocol is **memoryless**, if the final state of the agent FSMs is independent of the trajectory of traffic volumes over time, and only depends on the converged traffic volumes.

In particular, if we can verify that these properties hold for a given distributed system of agents running FSMs which implement flow reconstruction over a traffic-volume based LDS, then we know that the state of the FSMS depends only on the *converged* steady state of the network, and on the historical evolution leading to the steady state. Knowing

this allows us to predict system behavior from the converged traffic volumes, without having to run packet-level historical simulations. The resulting speed-up is essential to our success in analyzing the performance measures (see Section 4.4) dependencies on system parameters (see Section 4.3), and for making design and optimization problems (see Section 7.1) feasible.

Definition 4.4.9 already provides a formal rendering of the notion of a stable protocol. In what follows, we make precise the notion of a **memoryless** protocol.

4.5.1 Memorylessness

Definition 4.5.1. *If c eventually stable traffic history, we define $\overline{c}(V) = (\overline{c}(v)|v \in V)$ be the* ***collective traffic vector****.*

Definition 4.5.2. *If a protocol is eventually stable and c is an eventually stable traffic history, we define $\overline{\sigma}(V) = (\overline{\sigma}(v)|v \in V)$ be the corresponding* ***collective state vector****.*

Definition 4.5.3. *A protocol is* ***memoryless*** *if for any eventually stable traffic histories c_1 and c_2,*

$$\overline{c}_1(V) \equiv \overline{c}_2(V) \Rightarrow \overline{\sigma}_1(V) \equiv \overline{\sigma}_2(V).$$

In other words, if for any two eventually stable traffic histories with identical collective traffic vectors, the resulting collective state vectors are always identical, then the protocol is said to be memoryless. Informally, memoryless means that the final state of the system depends on its converged traffic volumes, not on the trajectory that the system takes to come to its converged traffic volumes.

4.5.2 Eliminating state variables

We have defined what it means for a protocol to be stable and memoryless. Note that this definition does not assume any state variables in the protocol. Because, our agent finite state machine FSM_8 has two variables, *ttl* and $\#Parents$ the previous definitions cannot be applied directly[1]. We need to incorporate the variables into the definition of memoryless.

Rather then extending the definitions of stable and memoryless to include state variables we will re-express FSM_8 in a way that eliminates state variables, by encoding their values into the state's identity. Specifically, we will rewrite FSM_8 as an equivalent Infinite State Machine (ISM) without $\#Parents$ variable, and show the ISM is memoryless and stable. With a suitable notion of equivalence, this will allow us to conclude the original FSM_8 is also memoryless and stable. Let us begin by defining what we mean by the equivalence of two Mealy machine FSMs. Informally, we would like to equivalence of two FSMs to mean that any sequence of inputs/events fed into the first FSM produces an output/action sequence which is identical to the sequence produced when that input/event sequence is fed into the second FSM; and vice versa.

We denote the set of inputs/event and output/action that appear in the two FSMs as Σ, and let Σ^* be the Kleene closure of Σ, i.e. the set of all finite strings over alphabet Σ.

Definition 4.5.4. *Let F be a non-deterministic finite state machine (NFSM), together with a chosen initial state I, and the set of inputs/event and output/action is Σ.*

1. We can ignore *ttl* because it is a local variable which is decremented at certain time intervals and it only effects the speed the decisions are made. On the other hand, $\#Parents$ is both incremented and decremented as a result of messages received from other agents.

The F gives rise to a ***transfer*** *function*

$$f : \Sigma^* \to 2^{\Sigma}$$

where for $w = X_1, X_2, ...X_n \in \Sigma^*$, $f(w)$ is defined as follows by defining $J_i \subset S$ as:

- $J_0 = \{I\}$
- $s \in J_i \leftrightarrow \exists r \in J_{i-1}$ *and there is a transition from* $r \to s$ *on input* X_i
- $u \in f(w) \leftrightarrow u$ *is the output associated with a transition* $r \to s$ *labeled by* X_n*, where* $r \in J_{n-1}$*, and* $s \in J_n$*.*

We say that F is a ***deterministic*** *finite state machine (FSM) if* $\forall w \in \Sigma^*$, $|f(w)| = 1$.

Definition 4.5.5. *Given two $NFSMs$ F_1 and F_2 over state sets S_1, S_2 with respective start states $I_1 \in S_1$ and $I_2 \in S_2$ and I/O over the set Σ, let $f_1, f_2 : \Sigma^* \to 2^{\Sigma}$ be the corresponding transfer functions. We say $F_1 << F_2$ if $\forall w \in \Sigma^*$ $f_1(w) \subseteq f_2(w)$; in this case we say that F_2* ***subsumes*** *F_1. If $F_1 << F_2$ and $F_2 << F_1$ than we say F_1 and F_2 are* ***equivalent****, denoted by $F_1 \equiv F_2$.*

The above notion of NFSM equivalence ensures that any sequence of inputs/events fed into the first NFSM produces an output/action sequence which is identical to the sequence produced when that input/event sequence is fed into the second NFSM; and vice versa.

We are now ready to describe an infinite state finite state machine ISM which is equivalent to FSM_8, but has no state variable $\#Parents$. First we describe ISM; then we show $ISM \equiv FSM_8$.

Figure 4.5.2 shows the ISM corresponding to FSM_8. In order to build the ISM, we took the original eight state FSM and repeated its lower part ∞ many times. While repeating, we kept all incoming and outgoing links except the two self-loops corresponding to incoming AD and ND messages. In the ISM those AD and ND links result in vertical state transition changing the state of the ISM to the next or previous level instead of self looping like in FSM_8. We removed the $\#Parents$ variable when we convert FSM_8 to ISM, because the value of the variable is now encoded in the level of the ISM.

Definition 4.5.6. *The states of the ISM are taken as the set*

$$\begin{aligned} S_I \;=\; & \{State_{0,1}, State_{0,2}, State_{0,3}, State_{0,4}, \\ & State_{1,1'}, State_{1,2'}, State_{1,3'}, State_{1,4'}, \\ & State_{2,1'}, State_{2,2'}, State_{2,3'}, State_{2,4'}, \ldots, \\ & State_{n,1'}, State_{n,2'}, State_{n,3'}, State_{n,4'} \ldots\} \end{aligned}$$

We define the state mapping function θ used to map a pair consisting of a state Si in FSM_8 and a value of the $\#Parents = p$ variable, onto the states of ISM:

$$\theta(Si, p) = \begin{cases} State_{p,i} & p = 0 \\ State_{p,i'} & p > 0 \end{cases}$$

We define $\delta(State_{p,i}) = p$; delta returns the level number of a state in the ISM.

Since $\delta(\theta(Si, p)) = p$, we see that the value of variable $\#Parents$ in FSM_8 is encoded in the level number of each state in ISM. The I/O structure within any given level is identical to FSM_8.

We denote as $\Sigma = \{AH, BH, AD, ND, ADA, Timeout, \ldots\}$, the set of inputs/event and

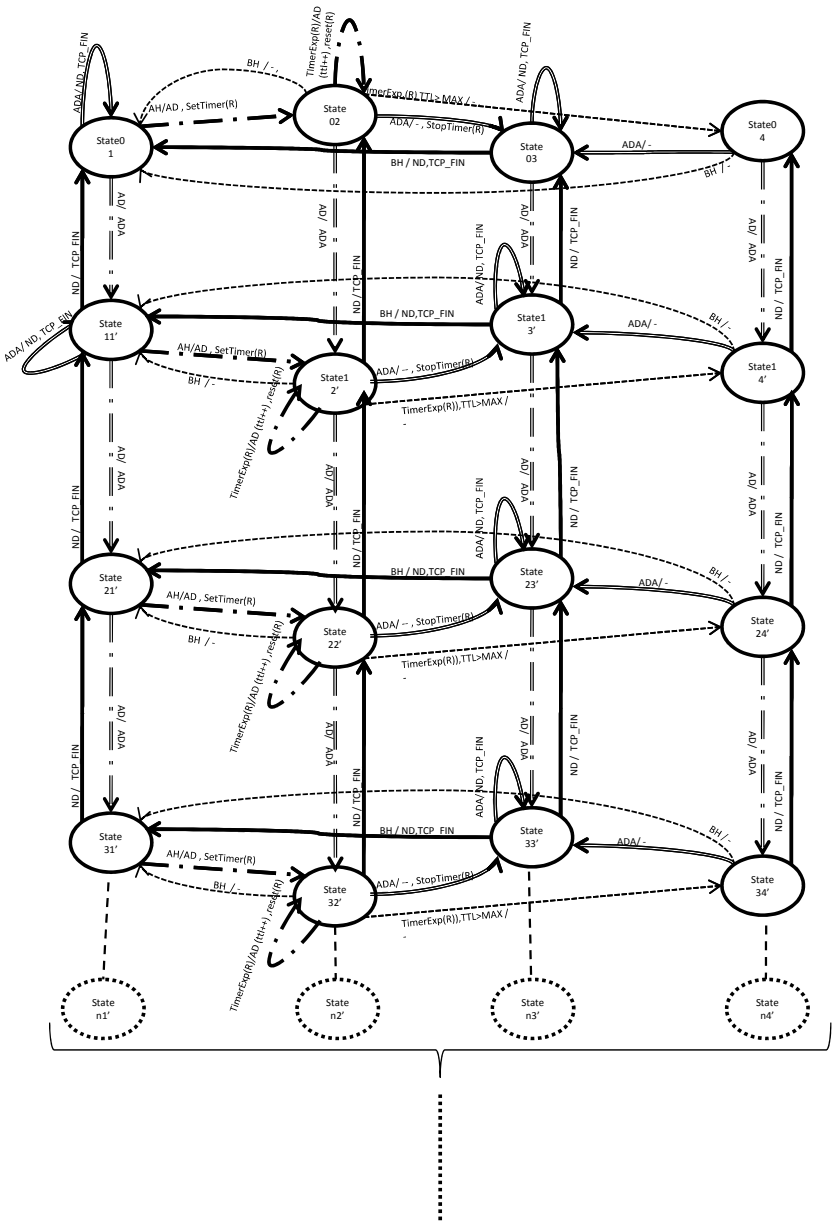


Figure 4.5.1: *ISM*

output/action that appear in the FSM_8. Let Σ^* be the Kleene closure of Σ; that is the set of all finite strings over alphabet Σ.

Lemma 1. *The ISM shown in Figure 4.5.2 is equivalent to* FSM_8

Proof. Assume ISM is not equivalent to FSM_8, towards contradiction. This requires that Definition 4.5.5 does not hold for the system. It means either FSM_8 is not subsumed by ISM or ISM is not subsumed by FSM_8.

Assume FSM_8 is not subsumed by ISM that means $\exists w \in \Sigma^*$ such that $f_{FSM}(w) \notin f_{ISM}(w)$. This means that there is $w = x_1, x_2$ where $x_1, x_2 \in \Sigma$ and $f_{FSM}(x_1) \subseteq f_{ISM}(x_1)$ but $f_{FSM}(w) \notin f_{ISM}(w)$ and let S_{x_1} be the state of FSM_8 and n_{x_1} be the value of $\#Parents$ after receiving x_1 respectively. If $f_{FSM}(x_1) \subseteq f_{ISM}(x_1)$ but $f_{FSM}(w) \notin f_{ISM}(w)$ then it means FSM_8 is in state S_{x_1} and has n_{x_1} parents at the same time it has an outgoing link corresponding to x_2 that changes FSM_8's state but ISM does not have such link. But we know that ISM is in state $\Theta(S_{x_1}, n_{x_1})$ that corresponds to S_{x_1}, n_{x_1} in FSM_8 with all incoming and outgoing links where the AD and ND messages does not self loop but change ISM's level. This means if there is an outgoing link at FSM_8 in S_{x_1}, n_{x_1} such that it changes the state of FSM_8 to S_{x_2}, n_{x_1} or to $S_{x_1}, n_{x_1} + / - 1$ then there is an outgoing link at ISM in state $\Theta(S_{x_1}, n_{x_1})$ such that it changes the state of ISM to $\Theta(S_{x_2}, n_{x_1})$ or to $\Theta(S_{x_1}, n_{x_1} + / - 1)$. Then there is a contradiction. Thus $FSM_8 << ISM$.

Now assume that ISM is not subsumed by FSM_8 that means $\exists w \in \Sigma^*$ such that $f_{ISM}(w) \notin f_{FSM}(w)$. This means that there is $w = x_1, x_2$ where $x_1, x_2 \in \Sigma$ and $f_{ISM}(x_1) \subseteq f_{FSM}(x_1)$ but $f_{ISM}(w) \notin f_{FSM}(w)$ and let $\Theta(S_{x_1}, n_{x_1})$ be the state of ISM after receiving x_1. If $f_{ISM}(x_1) \subseteq f_{FSM}(x_1)$ but $f_{ISM}(w) \notin f_{FSM}(w)$ then it means ISM has a state $\Theta(S_{x_1}, n_{x_1})$ that has an outgoing link corresponding to x_2 which changes ISM's state but FSM does not have such link. But we know that FSM is in

state S_{x_1} with number of parents n_{x_1} when ISM is in state $\Theta(S_{x_1}, n_{x_1})$ with all incoming and outgoing links where AD and ND messages are self loops. This means if there is an outgoing link at ISM in state $\Theta(S_{x_1}, n_{x_1})$ then there is an outgoing link at FSM_8 in S_{x_1}, n_{x_1}. Then there is a contradiction. Thus $ISM << FSM_8$

Since $ISM << FSM_8$ and $FSM_8 << ISM$, from Definition 4.5.5 $FSM_8 \equiv ISM$. □

4.5.3 Splitting ISM into upper and lower parts

We will split our original ISM in to two new types of FSMs, ISM_U and ISM_L, which operate independently within each node. In the old system, in any given agent node, for each victim address, we had one instance of FSM_8 (or ISM) running. In the new system, in any given agent node, for each victim address, we will have one instance of ISM_U running for each upstream agent, and one instance of ISM_L running for the downstream agent.

After finding such two separate FSMs, we will prove the memorylessness of the set of FSMs within a single fork in the agent tree, and then extend to the system as a whole.

Figure 4.5.3 shows the ISM_U, which is used to talk to the parent agents. ISM_U is obtained by projecting the states in the ISM horizontally, such that all incoming and outgoing links are preserved and combined in one state. Let $\pi_u : S \to T$ be a function where S is the set of states in ISM and T is the set of states in ISM_U. Let (u, v) be an edge connecting states u and v in ISM and $E(ISM)$ be the set of all edges in ISM. The reader can easily verify from the figures that $\forall(u, v) \in E(ISM)$ labeled by I/O implies that $\exists(\pi_u(u), \pi_u(v)) \in E(ISM_U)$ labeled by I/O.

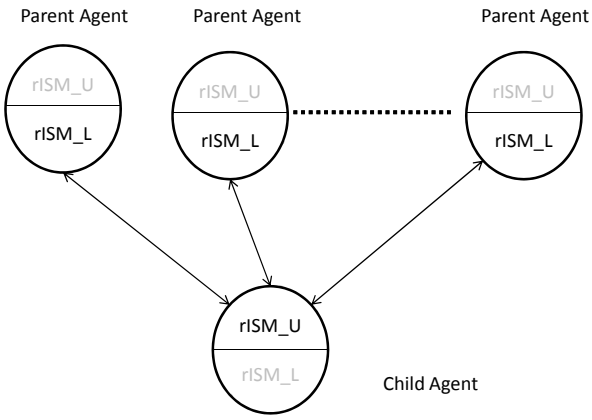


Figure 4.5.2: Split FSMs operating at a single fork in the agent tree.

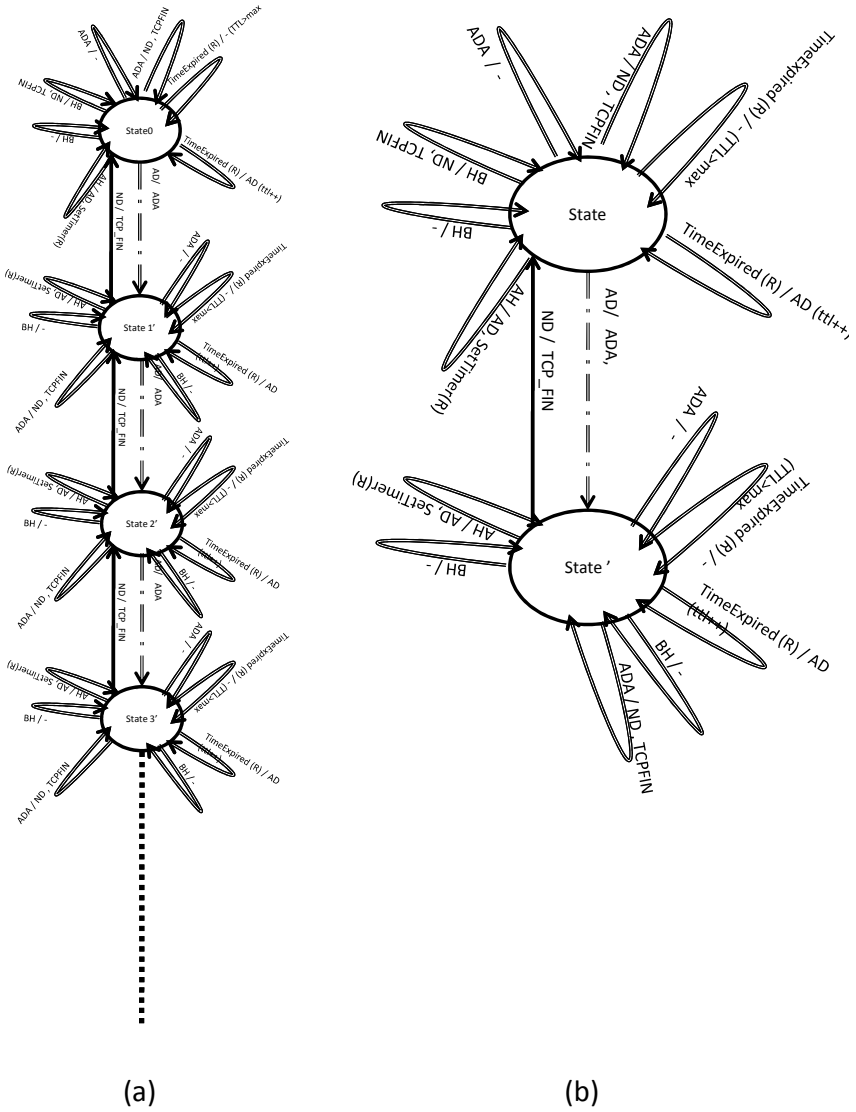


Figure 4.5.3: ISM_U (a)shows real ISM_U and (b) shows a bigger picture for readability

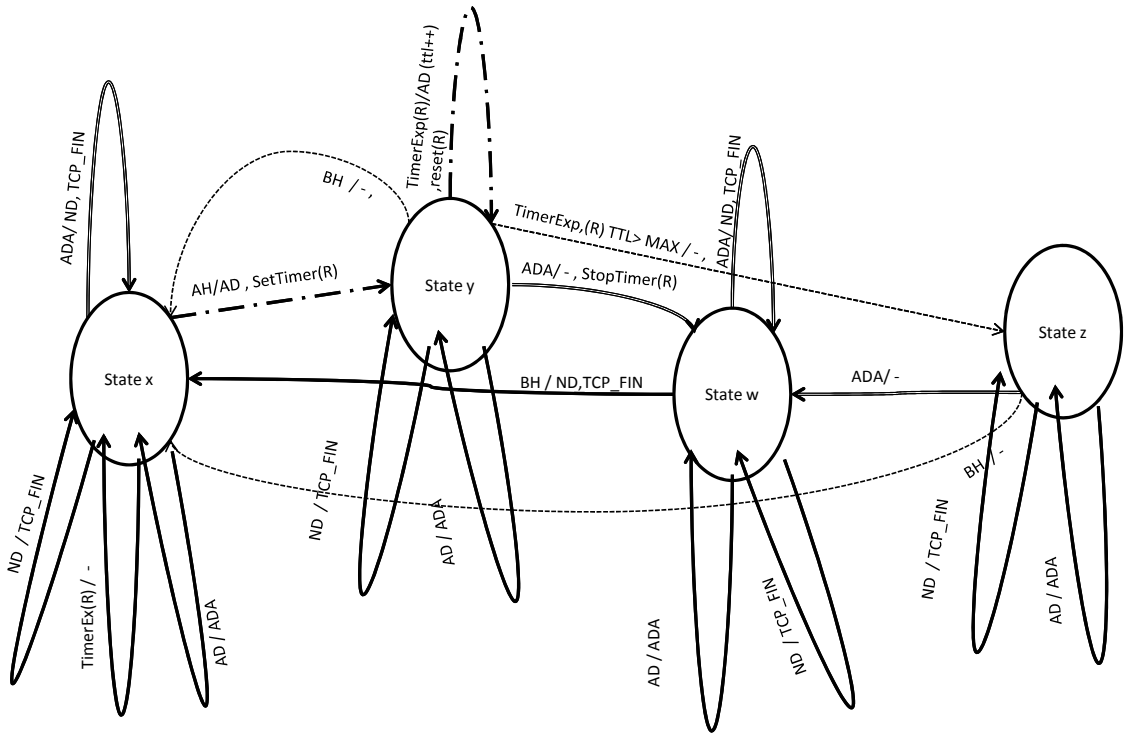


Figure 4.5.4: ISM_L

Figure 4.5.3 shows the ISM_L which is used to talk to the the child agents. ISM_L is acquired by projecting ISM vertically, such that all incoming and outgoing links are preserved and corresponding states in different levels are combined in a single state. Note that ISM_L is actually an FSM. Let $\pi_l : S \to T$ be a function where S is the set of states in ISM and T is the set of states in ISM_L. Let (u, v) be an edge connecting states u and v in ISM and $E(ISM)$ be the set of all edges in ISM. The reader can easily verify from the figures that $\forall(u, v) \in E(ISM)$ labeled by I/O implies $\exists(\pi_l(u), \pi_u(v)) \in E(ISM_L)$ labeled by I/O.

Next, we show that the ISM is subsumed by both ISM_U and ISM_L.

Lemma 2. *$ISM << ISM_L$ and $ISM << ISM_U$.*

Proof. Assume $X_1, X_2, X_3, ..., X_n$ is a sequence of messages accepted by ISM. Then it is also accepted by both ISM_U and ISM_L because there exists an edge in ISM for all accepted messages and we know that there is a corresponding edge in both ISM_U and ISM_L. This all messages accepted by ISM are also accepted by ISM_U and ISM_L, and produce the same output. □

4.5.4 Reducing the upper and lower FSMs

Now lets focus on Figure 4.5.3 where we can see ISM_U, and Figure 4.5.3 where we can see ISM_L. We begin by noticing that all arcs representing messages from below, are self loops in upper FSM; likewise all arcs representing messages from above, are self loops in lower FSM.

Note that ISM_U has several self-loop edges. Note also that some of them are dealing with messages coming from child agents and corresponding replies to them. If we assume ISM_U is used by the agent only for dealing with communicating with parents, we can remove these self-loop edges from ISM_U, since ISM_U will not receive any messages from children.

Note that ISM_L has similar self-loop edges. Here also, the self-loop edges are dealing with messages coming from agents parents and corresponding replies to the them. If we assume ISM_L is used by the agent only for dealing with communicating with the child, we can remove these self-loop edges from ISM_L, since ISM_L will not receive any messages from parents.

Note also that there are some other self-loop edges in both $ISMs$ which does not produce

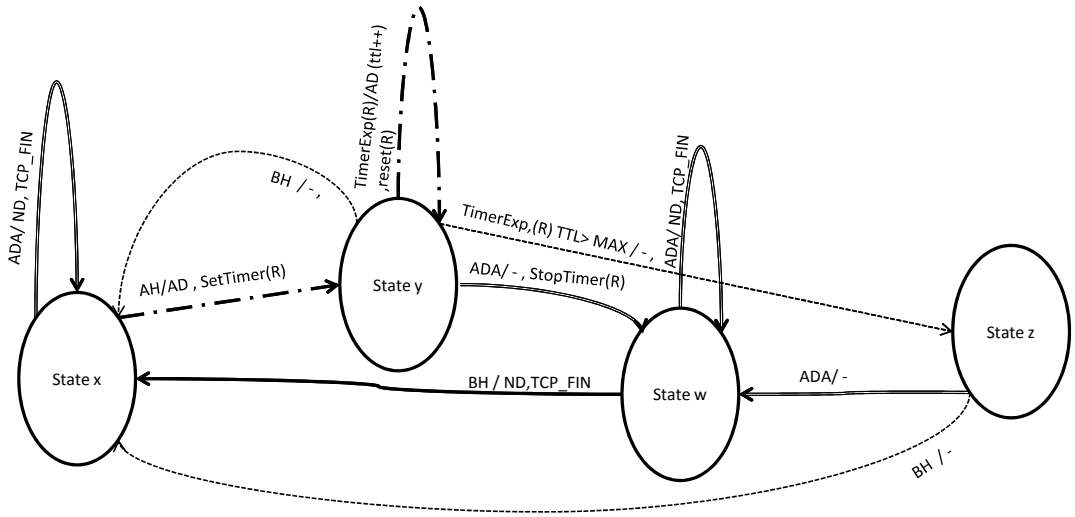


Figure 4.5.5: $rISM_L$

any outgoing messages. We can remove such edges too. The result of these removals are two reduce ISMs, which we call $rISM_L$ and $rISM_U$, respectively. Figure 4.5.4 shows $rISM_L$ and Figure 4.5.4 shows $rFSM_U$.

The aforementioned construction process is a direct argument for the assertion that by simultaneously operating of one $rISM_L$ and one $rISM_U$ (per parent) within an agent node, under the assumption that ISM_U is used only for communicating with parents and ISM_L is used only for communicating with child, gives rise to a system in which the simultaneous action of these FSMs is equivalent to the original ISM.

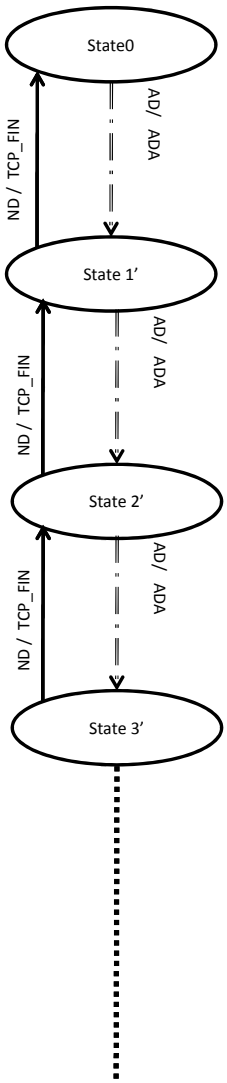


Figure 4.5.6: $rISM_U$

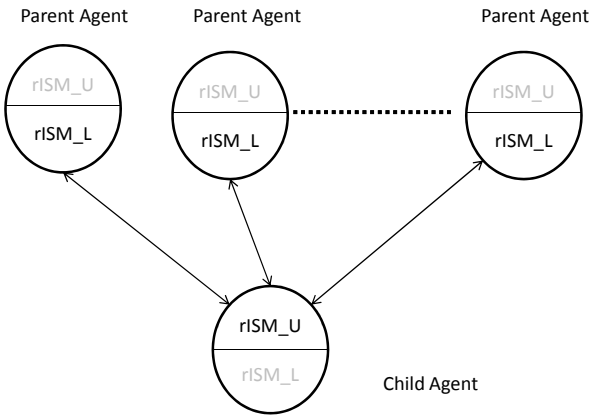


Figure 4.5.7: Attack Tree SubGraph (ATSG)

Our protocol requires an agent to have separate FSMs for each destination. Forany given victim, each agent runs one instance of $rISM_U$ when talking to each of its parents, and an instance of $rISM_L$ when talking to its (necessarily onlY) child. Figure 4.5.3 shows the inter-agent communication and associated reduced $ISMs$. This figure shows the topology between a single agent and its parents, relative to a fixed victim. Clearly, an agent can have multiple parents; it can have only one child for a specific attack flow because attack paths follow the physical routes depending on the routing tables of routers forwarding attack traffic.

In this section we will show that the FSM_8 is stable and memoryless. This is the same as showing that the ISM is stable and memoryless. This in turn requires that the $rISM_L$ and $rISM_U$ are stable and memoryless. Although there are many parents and associated $rISM_L$ instances in the figure, no parent's $rISM_L$ is effected by any of its siblings because there is only one link connected to it, and that is to the child–but no child agent ever initiates any communication to a parent. Each child agent runs an instance of $rISM_U$ when it talks to its parents. As seen in Figure 4.5.4, $rISM_U$ simply replies with ADA whenever it receives an incoming AD message. Thus, there is no information leakage at the child agent about its other parents, and consequently, a parent agent talks to its child in a way that is not affected by the child's other parents. Thus, it is enough to show that $rISM_L$ and $rISM_U$ are each stable and memoryless.

4.5.5 The $rISM_L$ is stable

Recall that $c(v,t)$ is the traffic at v at time t and $t_c = t_c(v)$ is the time at which the traffic at v stabilizes at volume $c(v,t_c)$. Let Γ be the maximum latency in G for an AD message to go from a parent agent to the child agent, and let δ be any positive infinitesimal.

Let $\Sigma_L = \{AH, BH, ADA, timeout\}$ be the set of messages accepted by $rISM_L$ and $S_L = \{\text{State } x, \text{State } y, \text{State } w, \text{State } x\}$ be the set of states that $rISM_L$ can be in. Alert AH is generated when $c(v,t) > threshold$ and $c(v,t-\delta) < threshold$. Alert BH is generated when $c(v,t) < threshold$ and $c(v,t-\delta) > threshold$. The ADA message is a message received from child agents, and the *timeout* message is created if no ADA is received when the timeout period expires.

Lemma 3. *No AH is produced at v after t if $t > t_c(v)$ and $c(v,t) < threshold$.*

Proof. AH is produced by v at t when $c(v,t) > threshold$ and $c(v,t-\delta) < threshold$. Since $c(v,t) < threshold$ and the traffic has already stabilized, there will be no future AH. □

Lemma 4. *No BH is produced at v after t if $t > t_c(v)$ and $c(v,t) > threshold$.*

Proof. BH is produced by v at t when $c(v,t) < threshold$ and $c(v,t-\delta) > threshold$. Since $c(v,t) > threshold$ and the traffic has already stabilized, there will be no future BH. □

Lemma 5. *If $t > t_c(v)$ and $c(v,t) < threshold$ then $rISM_L$ has converged to State x.*

Proof. If $\{t \mid c(v,t_i) > threshold\} \neq \emptyset$, let t_i be the largest element. Then there is a t_0 such that $t_i < t_0 < t$ and $c(v,t_0-\delta) > threshold$ and $c(v,t_0+\delta) < threshold$. The possible set of states that $rISM_L$ could be in at t_i is $S_L(t_i) = \{\text{State } y, \text{State } w, \text{State } x\}$. By the design of the protocol at t_0 a BH message is received by the agent. A BH message changes the state of the system from $\sigma(v,ti) \in S_L(t_i)$ into $\sigma(v,ti+\delta) = \text{State } x$. An agent leaves State x if and only if it gets an AH message. By Lemma 3, we know that there will be no later AH messages generated, so the FSM has converged to State x.

If $\{t \mid c(v,t_i) > threshold\} = \emptyset$, then no AH message was ever produced. Since every agent starts at State x and the only message that can change the agent's state from State x into other states is an AH, the agent has remained in State x for the entire duration. By Lemma 3, since the traffic is stable there will be no later AH message generated, which means the agent has stabilized to State x. □

Lemma 6. *If $t > t_c(v)$ and $c(v,t) > threshold$ an $rISM_L$ in State y without children, will end up in State z.*

Proof. The ADA message is produced by a child agent. Since there is no child, there will be no ADA message and the timer will expire at time $t_f = t+timer$ causing a *timeout* message to be produced. From the design of the protocol the *timeout* message causes a state change to State z. The normal set of messages accepted at State z are $\Sigma_z = \{ADA, BH\}$ but from Lemma 4 we know that there will be no BH arriving, and from our assumption of no child, we know that there will be no ADA. Consequently, Σ_z becomes $\Sigma_z = \{\}$. Since there will be no messages arriving, the system will stay stable in State z (after having passed through State y). □

Lemma 7. *If $t > t_c(v)$ and $c(v,t) > threshold$ an $rISM_L$ in State y with at least one child, will end up in State w.*

Proof. The ADA is produced by a child agent. Since there is a child there will be an ADA message then $\Sigma_y = \{ADA, timeout\}$. There are two cases. The system either gets *timeout* first and then ADA or the ADA arrives in time.

If *timeout* comes first, the state will change to State z. Since timer will expire at time $t_f = t + timer$, a *timeout* message will be produced. From the design of the protocol the *timeout* message changes the FSM to State z. The normal set of messages accepted at State z is $\Sigma_z = \{ADA, BH\}$ but from Lemma 4 we know that there will be no later BH, and from our child assumption we know that an ADA message will be generated by a child. Consequently, Σ_z becomes $\Sigma_z = \{ADA\}$. From the design of the protocol the ADA message changes state from State z to State w. The normal set of messages accepted at State w is $\Sigma_w = \{ADA, BH\}$ but from Lemmas 3 and 4, we know that there will be no BH or AH arriving. Consequently, $\Sigma_z = \{ADA\}$. From the design of the protocol the ADA message is a self-loop in State w, so the system will be stable in State w.

If ADA comes first, from the design of the protocol the ADA message causes a transition

to State w. The normal set of messages accepted at State w is $\Sigma_w = \{ADA, BH\}$ but from Lemma 4 we know that there will be no BH and from our child assumption we know that there may be an ADA. Consequently, $\Sigma_z = \{ADA\}$. From the design of the protocol the ADA message is a self-loop or State w so the system will be stable in State w. □

Lemma 8. *If $t > t_c(v)$ and $c(v,t) > threshold$ then $rISM_L$ is stable either in State w or State z.*

Proof. If $\{t \mid c(v,t_i) > threshold\} \neq \emptyset$, let t_i be the largest element. Then there is a t_0 such that $t_i < t_0 < t$ $c(v, t_0 - \delta) < threshold$ and $c(v, t_0 + \delta) > threshold$. The possible set of states that $rISM_L$ can be in at t_i is $S_L(t_i) = \{\text{State } x\}$. From the design of the protocol at t_0 an AH message is received by the agent. The AH message changes the state of the FSM $\sigma(v, t_0 + \delta) = \text{State } y$. When AH is received, a timer is started. From Lemma 4 we know that there will be no later BH, so $\Sigma_y = \{ADA, timeout\}$ is the set of messages accepted at State y. From Lemma 7 and Lemma 6 we conclude that $rISM_L$ is stable either in State w or State z. □

From Lemmas 5 and 8, we get:

Proposition 5. *If $t > t_c(v)$ then $rISM_L$ is stable at v at time t.*

4.5.6 The $rISM_U$ is stable

Let $\Sigma_U = \{AD, ND\}$ be the set of messages accepted by $rISM_U$ and

$$S_L = \{\text{State } 0, \text{State } 1', \text{State } 2', \text{State } 3'....\}$$

be the set of states that $rISM_U$ can be in. Note that AD and ND messages are received from parent agents. From the design of our protocol an AD message is sent down towards child agents every time an agent receives an AH message and a ND message is sent down towards child agents every time an agent receives an BH message. The protocol design enforces that an AD message changes the system state to a state one level up (i.e from State x to State $(x+1)$) and ND message changes the system state to a state one level down (i.e from State y to State $(y-1)$) where $x, y > 0$.

Lemma 9. *The first message received by an $rISM_U$ from a parent is an AD.*

Proof. An agent become a parent of another agent when it sends an AD message and receives an ADA message reply from the child. The design of our protocol mandates that AD message is sent over a connectionless protocol and ND is send using reliable TCP protocol. Assume, towards contradiction, that the first message received from a parent is ND. Which requires an established connection and connection is established with AD message. Then there must be an AD message before the ND message but since we assume that the first message received from a parent is ND we have a contradiction. As a result first message received from a parent can not be ND.

If an agent has a parent then it must have received a message from it, otherwise there will be no child-parent relation. We know that the set of messages that $rISM_U$ accepts is $\Sigma_U = \{AD, ND\}$. We saw that we need a message to create child-parent relation and we proved that the first message can not be an ND. It follows that the first message received from a parent is an AD message. □

Lemma 10. *No consecutive AH or BH are received an $rISM_U$.*

Proof. AH is produced by v at t when $c(v,t) > threshold$ where $c(v, t-\delta) < threshold$ and

BH is produces by v at t when $c(v,t) < threshold$ where $c(v,t-\delta) > threshold$. It follows from continuity assumptions on c that two AH message cannot be generated consecutively, nor can two BH messages. □

Lemma 11. *No consecutive AD or BD is received by an $rISM_U$ from same parent agent.*

Proof. Our protocol requires and agent to send an AD message whenever it gets an AH message and to send a ND message whenever it gets a BH message. From Lemma 10 we know that no consecutive AH or BH are received by an agent then no consecutive AD or BD will be sent to the child. As a result, no consecutive AD or BD is received by an child agent from same parent agent. □

Lemma 12. *If $\overline{c}(p) > threshold$, then the total number of AD messages received by $rISM_U$ from a parent agent p is one greater than the number of ND messages received from p.*

Proof. Recall $\overline{c}(p) = c(v, t_c(p))$. If $\overline{c}(p) > threshold$, then p did not receive any BH after the last AH it received at time t_c. As a result of AH p must send an AD to its child agent. Since there is no BH message after AH, p will not send any ND after last AD, and the last message received by child agent from p will be AD. From Lemma 9 we know that the first message received from p is AD, and from Lemma 11 we know that there is no consecutive AD or ND which means an AD is followed either by an AD or no message at all. Since the first and the last message is AD and no consecutive ADs or NDs are allowed, there must be $n-1$ ND messages for separating n distinct AD messages. □

Lemma 13. *If $\overline{c}(p) > threshold$, then the total number of AD messages received by $rISM_U$ from a parent agent p is one greater than the number of ND messages from p.*

Proof. Recall $\overline{c}(p) = c(v, t_c(p))$. If $\overline{c}(p) > threshold$, then p did not receive any BH after the last AH it received at time t_c. As a result of AH p must send an AD to its child agent.

Since there is no BH message after AH, p will not send any ND after last AD, so the last message received by child agent from p is AD. From Lemma 9 we know that the first message received from p is AD, and from Lemma 11 we know that there is no consecutive AD or ND which means an AD is followed either by an AD or no message at all. Since the first and the last message is AD and no consecutive ADs or NDs are allowed, there must be $n-1$ ND messages for separating n distinct AD messages. □

Lemma 14. *If $\overline{c}(p) < threshold$, then the total number of ND messages received by $rISM_U$ from a parent agent p is equal to the number of AD messages from p.*

Proof. If $\overline{c}(p) < threshold$ then p did not receive any AH after the last BH it received at time t_c. As a result of BH, p must send an ND to its child agent. Since there is no AH message after BH p will not send any AD after last ND then the last message received by child agent from p is ND. From Lemma 9 we know that the first message received from p is AD, and from Lemma 11 we know that there is no consecutive AD or ND which means an AD is followed either by an AD or no message at all. Since the first is AD and the last message is ND and no consecutive ADs or NDs are allowed, there must be $n-1$ ND messages for separating n AD messages and a last one that makes the total number of ND messages n. □

Proposition 6. *If $t > t_c(v) + 2\Gamma$ then $rISM_U$ is stable at v at time t.*

Proof. Recall that $\Sigma_U = \{AD, ND\}$. Let L_u be the superset of Σ_U, $P_U = \{p_1, p_2, ..., p_n\}$ be the set of parents, $p_i, p_j \in P_U$, $n(AH) = \#p_i$ where $c(p_i, t_c) > threshold$, and $n(BH) = \#p_j$ where $c(p_j, t_c) < threshold$.

Assume $L_u(t)$ is the language accepted by $rISM_U$ at time t when $c(V, t_c)$ is stable and $t > t_c + 2\Gamma$. Then by Lemma 14 and Lemma 13, $\#AD = \#ND + n(AH)$ in $L_u(t)$.

By design of the the protocol we know that the AD message changes the system state to a state one level up (i.e from State x to State $(x+1)$) and ND message changes the system state to a state one level down (i.e from State y to State $(y-1)$) where $y > 0$. Thus, by time t the state of $rISM_U$ has changed to level $(AD - ND)$ from its initial state. Since the initial state of the child is State 0, the final state of the child is State $_{0,(AD-ND)}$ = State $_{0,(ND+n(AH)-ND)}$ = State $_{0,n(AH)}$. Since $t > t_c(v) + 2\Gamma$, there will be no AD or ND messages sent between the agents after time t, consequently the $rISM_U$ remains at this level, stabilized. □

4.5.7 Stable subtrees

Lemma 15. *An agent v running $rISM_L$ and $rISM_U$ is stable.*

Proof. Lemma 5 shows that each agent running $rISM_L$ is stable and Lemma 6 shows that each agent running $rISM_U$ is stable. Since both of them are stable and $rISM_L$ and $rISM_U$ are independent of each other the agent is stable. □

Lemma 16. *Agent Tree SubGraph(ATSG) is stable.*

Proof. The ATSG is composed of a child agent and n parent agents. The child agent runs $rISM_U$ and each parent agent runs their own $rISM_L$ specific to the attack. Lemma 5 shows that each agent running $rISM_L$ is stable and Lemma 6 shows that each agent running $rISM_U$ is stable. □

4.5.8 The $rISM_L$ is memoryless

We endeavor next to show that $rISM_L$ and $rISM_U$ are both memoryless. Of these, the first argument requires more work, and so we begin with that. Recall that Definition 4.5.3 defines a memoryless protocol as one in which the converged state vector is determined solely by converged traffic vectors, and not by traffic history:

$$\forall c_1, c_2 : V \times \mathbb{R}^+ \to \mathbb{R}^+ \Rightarrow \overline{c}_1(V) \equiv \overline{c}_2(V) \Longrightarrow \overline{\sigma}_1(V) \equiv \overline{\sigma}_2(V)$$

Since c is the traffic history and the effect of the traffic is reflected on the state only to the extent that traffic is above or below the protocol threshold, there are four situations to consider which may arise:

(i) $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} \neq \emptyset$ and $c(\alpha, t_c) > threshold$. In this case, we denote the final state converged FSM state as $\sigma_1(\alpha, t_c)$.

(ii) $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} = \emptyset$ and $c(\alpha, t_c) > threshold$. In this case, we denote the final state converged FSM state as $\sigma_2(\alpha, t_c)$.

(iii) $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} \neq \emptyset$ and $c(\alpha, t_c) < threshold$. In this case, we denote the final state converged fsm state as $\sigma_3(\alpha, t_c)$.

(iv) $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} = \emptyset$ and $c(\alpha, t_c) < threshold$. In this case, we denote the final state converged fsm state as $\sigma_4(\alpha, t_c)$.

To show $rISM_L$ is memoryless, we need $\sigma_1(\alpha, t_c) = \sigma_2(\alpha, t_c)$ and $\sigma_3(\alpha, t_c) = \sigma_4(\alpha, t_c)$. This is achieved in the next four Lemmas.

Lemma 17. *If* $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} = \emptyset$ *and* $c(\alpha, t_c) < threshold$*, then the* $rISM_L$ *satisfies* $\sigma_4(\alpha, t_c) =$ *State* x*.*

Proof. Lemma 17 states that if $t > t_c(v)$ and $c(v, t) < threshold$ then the $rISM_L$ is stable in State x. Since α represents an agent running $rISM_L$ and $\forall t_i < t_c, c(\alpha, t_i) < threshold$ and $c(\alpha, t_c) < threshold$, so by Lemma 17 $\forall t_i$, $\sigma_4(\alpha, t_i) = \sigma_4(\alpha, t_c) =$ State x. □

Lemma 18. *If* $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} \neq \emptyset$ *and* $c(\alpha, t_c) < threshold$*, then the* $rISM_L$ *satisfies* $\sigma_3(\alpha, t_c) =$ *State* x*.*

Proof. Let $L_3 \subseteq \Sigma_L$ be the language accepted by α upto time t_c. In case (iii), the traffic history has property $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} \neq \emptyset$ so take t to be the largest element in this set. So $c(\alpha, t - \delta) > threshold$, and $c(\alpha, t + \delta) < threshold$. Our protocol design mandates the creation of BH at time t. Let L_{3L} and L_{3H} be the two parts of L_3 when it is divided into two parts: L_{3L} represents the language accepted until t (not including t), and L_{3H} represents the language accepted after t (including t). Then the first message in L_{3H} is BH. From Lemma 10, we know that no consecutive AH or BH is received by any, agent so there is a last AH in L_{3L} after which there is no BH.

Let S_{3L} be the possible states of α accepting L_{3L}. Our protocol design requires that AH is accepted at State x, causing the state to change State y. Thus, State $y \in S_{3L}$. By Lemma 20 then State w, State $z \in S_{3L}$. As a result $S_{3L} = \{$State w, State z, State $y\}$.

Let S_{3H} be the possible states of α accepting L_{3H}. Recall that the first message in L_{3H} is BH. S_{3L} is the set of states that the system is in just before BH. Our protocol design requires that BH changes any state in S_{3L}, to State x. Since $c(\alpha, t + \delta) = \overline{c}(\alpha)$ there is no AH or BH after t, so the system will be stable in State s. Thus $S_{3H} = \{$State $x\} =$ $\sigma_3(\alpha, t_c)$. □

Lemma 19. *If* $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} = \emptyset$ *and* $c(\alpha, t_c) > threshold$*, then the* $rISM_L$ *satisfies* $\sigma_2(\alpha, t_c) = \{State\ w\}$ *if* α *has at least one child, and satisfies* $\sigma_2(\alpha, t_c) = \{State\ z\}$ *if* α *has no child.*

Proof. Since α represents an agent running $rISM_L$, by Lemma 20 requires $\forall t_i$, $\sigma_2(\alpha, t_i) = \sigma_2(\alpha, t_c) = \{\text{State } w\}$ if α has a child.
Lemma 18 implies that $\forall t_i\ \sigma_2(\alpha, t_i) = \sigma_2(\alpha, t_c) = \{\text{State } z\}$ if α has no child. □

Lemma 20. *If* $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} \neq \emptyset$ *and* $c(\alpha, t_c) > threshold$*, then the* $rISM_L$ *satisfies* $\sigma_1(\alpha, t_c) = \{State\ w\}$ *if* α *has at least one child, and satisfies* $\sigma_1(\alpha, t_c) = \{State\ z\}$ *if* α *has any no child.*

Proof. Let $L_1 \subseteq \Sigma_L$ be the language accepted by α upto time t_c. In case (i), the traffic history has the property $\{t_i \mid c(\alpha, t_i) < threshold,\ t_i < t_c\} \neq \emptyset$, so take t to be the largest element in this set. Then $c(\alpha, t-\delta) < threshold$ and $c(\alpha, t+\delta) > threshold$. Our protocol design requires the creation of AH at time t.

Let L_{1L} and L_{1H} be the parts of L_1 when we divide it into two pieces, where L_{1L} represents the language accepted until t (not including t) and L_{1H} represents the language accepted after t (including t). Then the first message in L_{1H} is AH. From Lemma 10, we know that no consecutive AH or BH events are received by an agent, so there is a last BH in L_{1L} after which there is no AH. Let S_{1L} be the possible states of α accepting an element in L_{1L}. Our protocol design requires that BH causes a state change to State x, so State $x \in S_{1L}$. Again from our protocol design, only an AH message can cause the system's state to change away from State x. Since we have shown no later AH arrives, $S_{1L} = \{\text{State } x\}$.

Let S_{1H} be the possible states of α accepting L_{1H}. Recall that the first message in L_{1H} is

AH. $S_{1L} = \{\text{State } x\}$ is the set of states that the system is in just before the AH arrives. Our protocol design requires that AH changes systems state from State x to State y. Since $c(\alpha, t+\delta) = \overline{c}(\alpha)$ there is no AH or BH arriving after t, so by Lemma 7 the $rISM_L$ will end up in State w if it has a child, and by Lemma 6 it will end up in State z if it has no child. □

Proposition 7. *$rISM_L$ is memoryless.*

Proof. Lemma 18 and Lemma 17 shows that $\sigma_3 = \sigma_4$ and Lemma 20 and Lemma 19 shows that $\sigma_1 = \sigma_2$ no matter if α has any child or not[2]. Thus $\sigma_1(\alpha, t_c) = \sigma_2(\alpha, t_c)$ and $\sigma_3(\alpha, t_c) = \sigma_4(\alpha, t_c)$. As a result $rISM_L$ depends only on the current stable traffic and it is independent of the traffic history. This completes the proof. □

4.5.9 The $rISM_U$ is memoryless

We endeavor next to show that $rISM_U$ is memoryless.

Proposition 8. *$rISM_U$ is memoryless.*

Proof. Recall that $\Sigma_U = \{AD, ND\}$ is the set of messages accepted by $rISM_U$, and $S_L = \{\text{State } 0, \text{State } 1', \text{State } 2', \text{State } 3'....\}$ is the set of states that $rISM_U$ can be in. Since AD and ND messages are received from parent agents, $\sigma(rISM_U, t)$ is independent from possible AH and BH messages which depend on the local traffic. Consequently, we need to consider only the parents of $rISM_U$ to show that it is memoryless.

2. Having child or not does not change the definition of memoryless because the existence of a child does not change the traffic history.

Definition 4.5.3 declares that a protocol is memoryless if:

$$\forall c_1, c_2 : V \times \mathbb{R}^+ \to \mathbb{R}^+ \Longrightarrow \overline{c}_1(V) \equiv \overline{c}_2(V) \Longrightarrow \overline{\sigma}_1(V) \equiv \overline{\sigma}_2(V)$$

Recall Definition 4.5.2, $\overline{c}(V) = (\overline{c}(v)|v \in V)$ and $\overline{c}(v) = c(v, t_c(v))$. Let $P_U = \{p_1, p_2, ..., p_n\}$ be the set of parents and $n(AH) = \#\{p_i \in P_U \mid c(p_i, t_c) > threshold\}$. Define $\overline{c}(P_U)$ as the sequence $(c(p, t_c(p) \mid p \in P_U)$. We know that parent agents run $rISM_L$ when they talk to their children, and by Proposition Lemma 7 their FSMs are memoryless, hence the value of $n(AH)$ is memoryless[3]. Let $L_u(t)$ be the language accepted by $rISM_U$ upto time t, for $t > t_c + (2x\Gamma)$. Then by Lemmas 14 and 13, for any word w in $L_u(t)$,

$$\#AD(w) = \#ND(w) + n(AH).$$

From the design of the protocol we know that an AD message changes the system state to one level higher, i.e from State x to State $(x + 1)$; ND messages change the system state to one level lower, i.e from State y to State $(y - 1)$, where $x \geqslant 0, y > 0$. It follows that by time t the state of a child's $rISM_U$ has undergone a level change of $\#AD(w) - \#ND(w)$ Since the initial state of the child is State 0 (i.e. level 0), the final state of the child is

$$\text{State } 0 + (\#AD - \#ND) = \text{State } 0 + (\#ND + n(AH) - \#ND) = \text{State } 0 + n(AH).$$

Since $n(AH)$ is memoryless, the final level at which the FSM stabilizes, is independent of traffic trajectory. Hence, $rISM_U$ is memoryless. □

3. In the sense that its converged value depends only on the stabilized traffic vector, and not on the traffic trajectory.

4.5.10 Memoryless subtrees

Proposition 9. *Agent Tree SubGraph(ATSG) is memoryless.*

Proof. The ATSG is composed of a child agent and n parent agents. The child agent runs $rISM_U$ and each parent agent runs their own $rISM_L$ specific to the attack on victim v. Proposition 7 shows that each $rISM_L$ is memoryless and Proposition 8 shows that each $rISM_U$ is memoryless. The ATSG is memoryless. □

Proposition 10. *An agent v running $rISM_L$ and $rISM_U$ is memoryless.*

Proof. Proposition 7 shows that each $rISM_L$ is memoryless and Proposition 8 shows that each $rISM_U$ is memoryless. Since $rISM_L$ and $rISM_U$ are independent of each other, the agent running both $rISM_L$ and $rISM_U$ consequently is memoryless. □

CHAPTER 5

SIMULATION DESIGN AND IMPLEMENTATION

5.1 Discrete event simulation framework

In order to do packet level simulations, we used the framework for discrete-event simulation (FDES) which is described in detail in the second chapter of the book named "Network Modeling and Simulation: A Practical Perspective" [44]. At the heart of any centralized discrete event simulator is a "Scheduler", which acts as the centralized focal point for the passage of time. The scheduler contains a linearly order set of events scheduled for the future, and is responsible for executing them sequentially and chronologically. An any given point in time, the event at the head of this list is being delivered "now". Events are defined to be directed interactions between two entities at a particular time. The entities in our simulation are represented in our code by a base abstract Java class called SimEnt. A SimEnt represents a unit of logic which responds to the arrival of Events. The SimEnts response can involve: (i) sending new Events to other SimEnts, (ii) creating new SimEnts, (iii) destroying existing SimEnts. The influence transmitted between one (source SimEnt) and another (target SimEnt) is embodied in Java classes implementing the Event interface. We note that this one-to-one directed model does not cover all types of influence. In particular, one-to-many and many-to-many influences are not covered directly. However, these more complex forms of interaction can be simulated using higher-level constructs built over the elementary one-to-one constructs. The timed delivery of Events between

SimEnts is mediated by the Scheduler, and drives the discrete forward passage of time.

5.2 Flow reconstruction problem(FRP) simulation code overview

5.3 Simulation code

In this section, I will introduce the details of the Java code used for simulation of my system. From here on I will call the code as "FRPCode". I used NetBeansIDE 6.7, a free open-source integrated development environment(IDE), as my programming environment.

5.3.1 Code design

Recall that input to FRP is a tuple that consists of G, R, S, D, v, and the corresponding output is an overlay network of agents. In order to successfully simulate the FRP problem and perform the measurements on the resulting overlay network, we must provide FRPCode with required data. Consequently, we start with creating G, R, A, D, v. For each simulation experiment, this input must be created first. However it is not always the case that we recreate all of the required input; depending on the goal of the experiment, most of the time we just change one parameter of the input and repeat the experiments. For example, in order to see the effects of different agent sets on a certain experiment case, we must keep G, R, D, v fixed and create a new A.

There are different kinds of classes used in "FRPCode":

- Abstract Factory Classes(AFC)
- Sub-Factory Classes(SFC)
- Product Classes(PC)

An **AFC** is a Java class file. Its purpose is to provide a unique interface to the rest of the FRPCode hiding the internal complexity of underlying code. An AFC can have several sub-factory classes(SFC)s as its child. The creation method of each member of G, R, A, D, v can change from experiment to experiment. AFCs provide easy ways to switch between different kinds of experimental needs. In each AFC, we defined different creation methods and corresponding links to their appropriate SFCs. Whenever requested, any AFC returns a reference to the appropriate SFC. Interfaces to the SFCs are defined by their parents' AFCs; consequently communication between sibling SFCs is achieved using the same set of methods. Whenever an AFC is created, it is given which SFC to use to do its required operations.

FRPCode includes following AFCs:

- Network Factory (Netf): Used to create a new network for the simulation
- Routing Table Factory (RTf): Used to create the routing table for a given network
- Agent Factory (Agentf): Used to create a set of agents for the given network
- Attacker Factory (Attackf): Used to create a set of attackers for the given network
- Victim Factory (Victimf): Used to create a victim for the given network
- FRP Factory (FRPf): Used to create a FRP

- Solution Factory (Solf): Used to create solution for the given FRP.
- Measurement Factory (Measurementf): Used to create measurements for the given solution

A **SFC** is a Java class file, whose purpose is to provide service to requesters with the interface defined by its parent AFC. Every SFC has different job and different complexity, however they look the same to other classes.

For example, there is an AFC named **Agentf** which has two SFCs, SimpleAgentf and GreedyAgentf respectively. In some of the simulation experiments, we need simple agent placement, and in some others we need greedy agent placement. In order to handle this kind of switching, all we need to do is change the agent creation setting from **Agentf.SIMPLE** to **Agentf.GREEDY** and keep the rest of the code same. Consequently, by changing one setting we are able to use either SimpleAgentf or GreedyAgentf to create new agents for any experiment.

FRPCode includes following SFCs for the specified AFCs:

- Netf
  - Waxman Network Factory: Creates desired number of vertices, and places them randomly in a 2D plane, then it creates links to connect vertices. Links are created with respect to their Waxman probability of existence that is inversely proportional to the Euclidean distances of the two vertices that the link connects. We described the details later in this chapter.
  - CAIDA Network Factory: Creates an instance of a real life Internet topology according to the adjacency data which is freely provided by CAIDA project.

  - From File Network Factory: Creates an instance of a network that was previously written to a file by the FRPcode.
  - Test Network Factory: Creates a grid like network for testing purposes.

- RTf
  - Dijkstra Routing Table Factory: Creates the routing table for the whole network using Dijkstra's shortest path algorithm.
  - On demand Dijkstra Routing Table Factory: Creates the routing table for a single destination whenever it is needed. This is used when we run experiments on CAIDA network. We use this because the creation of whole Dijkstra routing table for CAISA network takes to much time and memory. Consequently, we used on demand Dijkstra routing to create routes as we need them.

- Agentf
  - Simple Agent Factory: Creates a set of agents by randomly selecting the agents among the vertices in the network.
  - Greedy Agent Factory: Creates a set of agents by using greedy algorithms which are designed to optimize the final values of M1 or M3 values.
  - Testing Agent Factory: Creates agent set for testing net only.

- Attackf
  - Simple Attacker Factory : Creates a set of attackers by random selection.
  - Incremental Attacker Factory : Creates a set of attackers however, each new set is a superset of the previous set
  - Sequential Attacker Factory : Creates an attacker sequentially

- Victimf
  - Simple Victim Factory: Creates a victim by random selection among vertices
  - Sequential Victim Factory: Creates a victim by sequentially going over all the vertices
- Solf
  - Central Solution Factory: Produces centralized solution to the given FRP
  - Distributed Solution Factory: Produces distributed solution to the given FRP
- Measurementf
  - M1f Measurement Factory: Creates M1f values for given solution
  - M2f Measurement Factory: Creates M2f values for given solution
  - M3f Measurement Factory: Creates M3f values for given solution

A **PC** is a Java class file which is used for storing the instances of objects created by SFCs. Whenever an SFC is requested to create, the output it produces is stored as a PC object. For example, whenever simple agent factory is requested to create, it creates a random set of agents and returns it. The returned agents set is stored as a PC object. PC object is then easily be reached by other components of the FRPcode.

Figures 5.3.1, 5.3.2, and 5.3.3 show the components of FRPCode and their relationships. In figures, the boxes with diamond shape inside represent AFCs; the boxes with arrows toward AFCs represent SFCs and the rest of the boxes represent PCs.

There are different types of relations between the components. Small arrows between a pair of boxes point to the parent of the other box. For example, there is a small arrow

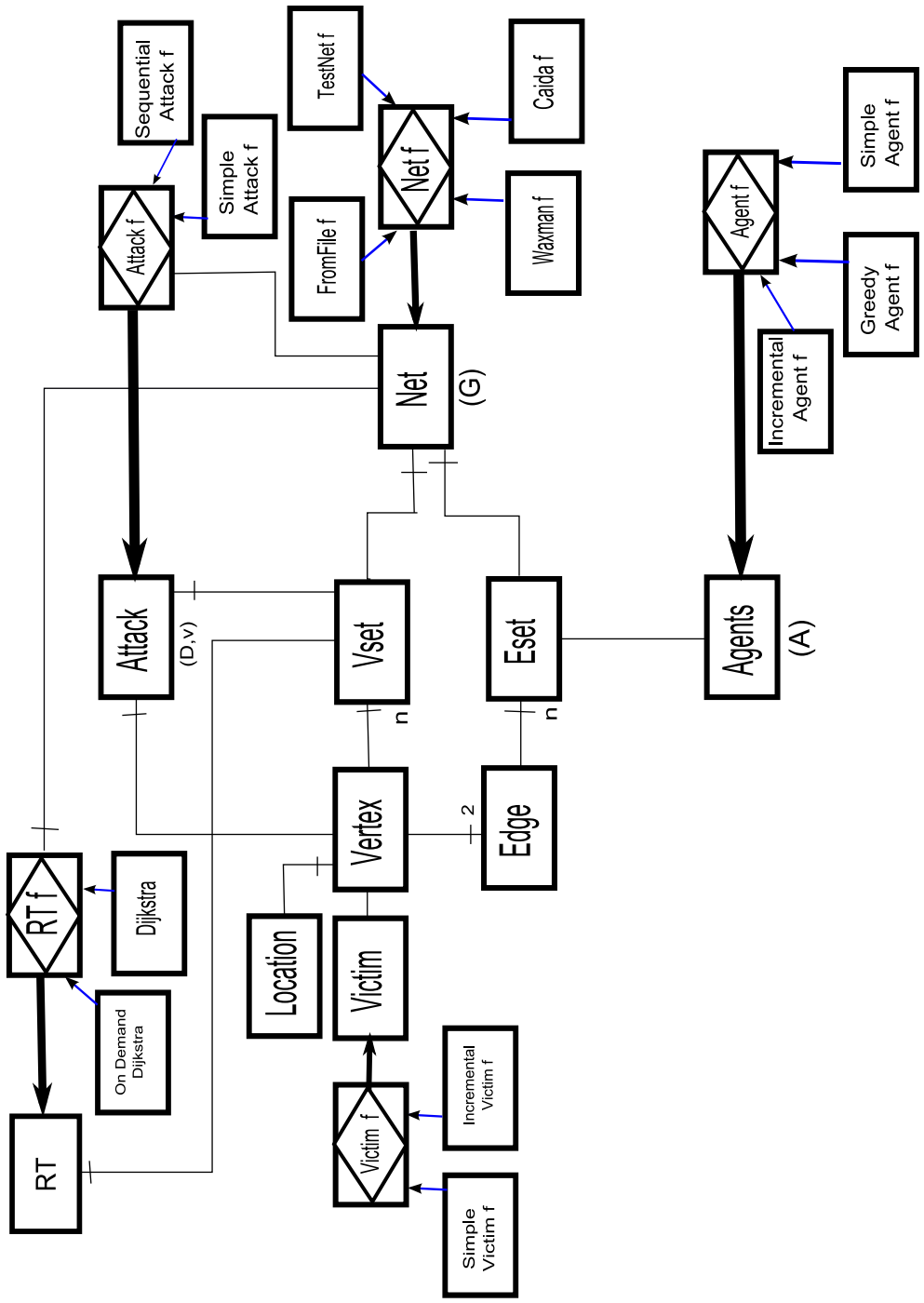


Figure 5.3.1: FRP code part one shows the structure and relationships of Netf, Agentf, Attackf, RTf, and Victimf

between the box labeled 'CAIDAf and 'Netf'. In this case, Netf is the parent of CAIDAf and CAIDAf is a SFC. A large arrow points to the product of the other box. For example, the large arrow between 'Attackf' and 'Attack' shows that 'Attackf' produces an 'Attack' which is a PC. A straight line shows that there is a relationship between two boxes and small perpendicular line on lines closer to one of the boxes means that this PC includes other PC as a data member. For example, there is a line between 'Net' and 'Vset'. This means 'Net' contains a 'Vset' as its data member. Numbers under perpendicular lines represent the number of instance of the other PCs included as data members in the other PC. If there is no number under the perpendicular line, than it corresponds to '1' and letter 'n' means any. For example, there is a line between 'Eset' and 'Edge' and perpendicular line closer to Eset with number 'n' underneath which means Eset can include any number of Edges.

In this section I will briefly describe the components of FRPcode in greater detail starting from the simple ones to complex ones.

5.3.1.1 Location

Usage: In our network G, nodes are located in a 2D plane. Each node must have a unique (x, y) coordinate. Location class provides unique location for each node. The Location class is used for keeping the location information related to a node. Whenever a new vertex object is created and a unique (x, y) coordinate is associated with it.

Location object provides two useful services, "Distance to a given node" and "Show locations" respectively. Distance to given node returns the Euclidean distance of those two nodes. Show location returns the (x, y) coordinate of the node.

Data Members:

xAxis is of type integer and stores the x-axis of the location object, **yAxis** is of type integer and stores the y-axis of the location object, **nodeLocations** is a static linked list that keeps the location information of all the nodes in the system so not to have more than one nodes at same coordinate, **distinct** is a boolean which is used to determine if a unique location is found.

Methods:

DistanceTo(Node location) method returns euclidean distance of this location to the given node, **locToInt()** converts the location information into an integer so that it can be placed in **nodelocations** list to provide uniqueness, and **showLocation()** prints out the location information on the screen.

5.3.1.2 Vertex

Usage: Vertices corresponds to network nodes. Vertex class is used to create vertex objects.

Data Members: The data members of the vertex class are **location**, **id**, and **name**. **Location** is an instance of location class and it keeps the location information of the vertex, **id** is an integer which represents a unique integer and the ID of the vertex, and name is a string which corresponds to the name of the vertex.

Methods: Vertex class has several methods. **CompareTo(Vertex)** is one of the methods and it is used to compare two vertices, and decide if two objects corresponds to the same vertex, **getid** method is used to get the vertex id , **getlocation** method is used to get the location of the vertex, **getName** method is used for retrieving the name of the vertex and

finally **toString** method is used to output readable information about the vertex.

5.3.1.3 Edge

Usage: Instances of Edge class correspond to the edges of graph G. Edge class is used to create edge objects. Each edge object keeps information about the end vertices, and the weight of the edge.

Data Members: The first data member of the Edge class is **id**. Id is used to store the unique id of the edge. The second data member is **src** which stores information about the vertex in one end of the link. The third data member is **dst** which stores information about the vertex in the other end of the edge, the final data member is **weight** which stores the weight information of the edge.

Methods: There are several methods that are provided by edge class. The **getDstVertex** method returns the vertex stored as dst vertex, **getSrcVetex** method returns the vertex stored as src vertex, **getWeight** method returns the weight of the link, and finally **toString** method returns a readable information about the edge object

5.3.1.4 Vset

Usage: Vset represents vertex set. Vset is used to store the set of vertices in G. It provides methods to retrieve a vertex by its name or by its id.

Data Members: The most important data member of Vset is a linked list named **vertexSet**. It is the structure that keeps the list of vertices in the network.

Methods: There are several methods. First, **addVertex(Vertex)** which is used to add a vertex to the list of vertices, **getRandomV** method is used for retrieving a random vertex from the set of vertices, **getVertexById(ID)** is used for given a vertex id returning the corresponding vertex, **getVertexByName(name)** is used to retrieve the vertex named **name**, **listMembers** method lists the members of **Vset**, **memberToFile** method writes the vertex information to a file, **removeVertex** method removes a vertex from the vertex set, and finally **toString** method converts the vset to a list of vertices.

5.3.1.5 Eset

Usage: Eset represents the set of Edges. Eset is used to store the set of edges in G. It provides methods to retrieve a vertex by its name or by its id.

Data Members: The most important data member of Eset is a linked list name **edgeSet**. It is the structure that keeps the list of edges in the network.

Methods: There are several methods. First, **addEdge(source,destination,weight)** which is used to create and add an edge to the list of edges given source vertex, destination vertex and weight, **getRandomEdge** method is used for retrieving a random edge from the set of edges, **getEdgeBetweenVertices(Vertex1,Vertex2)** returns either the corresponding edge for the given two vertices or null if edge does not exist between the given vertices, **memberToFile** method writes down the information about the set of edges, **listMembers** method lists the members of **Eset**, **removeEdge** method removes an edge from the vertex set, **toString** method converts the Eset to a list of edges, and finall **doesEdgeExists(Vertex1,Vertex2)** method checks if there is an edge between the given vertices, and returns true or false.

5.3.1.6 Victim

Usage: Victim class is used to store information about a victim. Any victim is a vertex object. The victim class has a unique victim identifier.

Data Members: Victim class has two data members, **victimID** and **victim**. **victimId** is an integer and assigns a unique integer number to the victim as it is created, **victim** is an instance of a vertex and it stores the vertex information of the victim.

Methods: Victim class has several methods; **getId**, **membersToFile**, and **victimToVertex**. **getId** returns the id number of the victim, **membersToFile** writes the victim information to an external file, and **victimToVertex** method returns the vertex corresponding to the victim.

5.3.1.7 Agents

Usage: Agents class is used to store the set of agents in G. Agents are located on edges. Consequently, agents set keeps the list of edges on which agents are installed.

Data Members: Agents class has two data members, **AgentSetID**, and **AgentsSet** list respectively. **AgentsSetId** is a unique integer represents the ID of the agents set and **Agentset** is a linked list which keeps record about all the agents in G.

Methods: Besides common methods like **listMemebers** and **membersToFile**, the Agents class has a method named **getAgent(Vertex1, Vertex2)**. **GetAgent** method returns the corresponding agent given the information about two ends of a link. If there is an agent on the link then **getAgent** method returns the edge on which the agent is placed, if there

in no agent then the method returns null.

5.3.1.8 Attack

Usage: Attack class is used to store the set of attackers in G. Attackers are vertices. Consequently, attack class keeps the list of vertices which are the attackers attacking to other systems.

Data Members: Attack class has two data members namely **atackerSetID** and **attackers**. The attackerSetId is an integer that keeps the record of unique attackers set ID and attackers is a hashset which keeps records about the attackers.

Methods: Besides trivial set and get methods, there is a new method named **isAtacker** which returns true if the given vertex is an attacker and returns false if the given vertex is not in attackers set.

5.3.1.9 Net

Usage: Net class keeps information about the edges and the vertices of G. It keeps the list of vertices, and edges in the network. Moreover, it keeps some additional useful information regarding the network like the number of links, the number of nodes in the network, and if the net is connected.

Data Members: The data member named **netID** is an integer that stores the ID number of the current network, **vertexSet** and **EdgeSet** are being used for storing information about the vertex set and edge set, **neighbours** is a HashMap that keeps the set

of vertices adjacent to the key vertex, **numberOfNeighbours** is another HashMap that stores the number of neighbors for each vertex in the network, **vertexWithMaxNumOfNeighbours** is an instance of vertex which has the maximum number of neighbors, **maxNumOfNeighbours** is an integer that keeps the maximum number of links and **NumberOfLinks** is another integer which keeps the number of nodes in the network.

Methods: The Net class has several methods. **AddEdge** method adds a given edge to the net object, **addVertex** method similarly adds a given vertex to the net, **checkDijkstraCompletemethod** is a private method to check if the network is fully connected or not, **getRandomEdges(s)** method returns a set of edges with size s, **getRandomV** method returns a randomly selected vertex, **getRandomVertices(s)** method returns a set of vertices with size s, **getVertexWithOneNeighbour** method returns a vertex that has only one neighbor, **listEdges** method lists all the edges in the network, **listVertices** method lists all the vertices in the network, and **removeVertexFromNet**(vertex) method removes the given vertex from the network.

5.3.1.10 Victimf

Usage: The purpose of **Victimf** is to provide the simulation with a unique interface for creating victims. **Victimf** has two children namely, **simpleVictimf** and **sequentialVictimf**. There is a static method called **makevictimf** in Victimf. Whenever makeVictimf method is called with inputs network G and victimf type, the victimf class creates a new victimf object and returns it. If the requested victim creation type is "Simple" then a handle to simple victim factory is returned; similarly if the requested type is "Sequential" than a handle to sequential victim factory is returned and so on.

Data Members: Victimf defines two types namely, SIMPLE and SEQUENTIAL.

Methods: Victimf has only one abstract method called create method. All child classes must have create method which creates a victim for the experiment and returns it to the requesting object.

5.3.1.11 SimpleVictimf

Usage: SimpleVictimf stands for simple victim factory. Its purpose is to randomly create a victim and return the result.

Data Members: It has only one data member **net**. Net is an instance of Net class.

Methods: It has only one method named **create**. Create method selects a random vertex as the victim and returns it.

5.3.1.12 SequentialVictimf

Usage: SequentialVictimf represents sequential victim factory. Its purpose is to select the next vertex as victim by sequentially returning the next vertex whenever a new victim is requested. This is used when we want to see the affect of attacking all the vertices sequentially.

Data Members: It has three data members; namely **net**, **remainingVertexSet**, and **tmpVertexSet**. Net is a reference to the current network, remainingVertexSet is the list of all the vertices, and tmpVertexSet is the list of vertices which are not selected to be victim yet.

Methods: It has only one method named **create**. Create method selects the next vertex from tmpVertexSet and returns it as the victim.

5.3.1.13 Netf

Usage: Netf stands for network factory. The purpose of **Netf** class is to provide the simulation with a unique interface for creating the network. **Netf** has four children namely, **Waxmanf**, **CAIDAf**, **TestNetf**, and **FromFilef**. There is a static method called **makeNetf**. Whenever makeNetf method is called with network G, number of nodes, edge density, area and netf type as input, the netf class creates a new netf object and returns the link to the object. If requested network creation type is Waxman then a handle to waxmanf is returned similarly if the requested type is Caida then a handle to Caidaf is returned and so on.

Data Members: Netf defines four network creation types:WAXMAN, CAIDA, TESTINGNET and NETFROMFILE.

Methods: Netf has only one abstract method called create method. All child classes must have create method which creates a network for the experiment and return it to the requesting object.

5.3.1.14 Waxmanf

Usage: Waxmanf stands for Waxman network factory that creates a random network according to Waxman [64] probabilities. The purpose of Waxmanf is to create a network in which node locations are randomly selected and links between nodes are created inversely

proportional to the exponential Euclidean distance of two nodes according to the formula $a * e^{\frac{-d}{bL}}$, where a and b are constants, d is the Euclidean distance of two nodes and L is the maximum node distance in the . Whenever Waxmanf is requested to create a new network, it starts creating the network by first creating the nodes and randomly selecting their (x,y) coordinates in a $5\sqrt{|V|} \times 5\sqrt{|V|}$ 2D plane. After creating the nodes it computes all possible links with their Waxman possibilities. After creating all possible links it places the links on a line such that a link with greater possibility occupies larger place and a link with smaller possibility occupies less place on the line. After lining up all possible links a random number is chosen and the link corresponding to that number is selected as next edge for the network being created. This selection is continued until the desired edge density is reached. Once the desired edge density is reached the network is checked if it is connected or not. If it is not connected when the edge density reaches 2.1, we continue add edges until the graph becomes connected.

Data Members: The data members of Waxmanf: edgeDensity, area, numOfNodes, WAXMANBeta, WAXMANAlfa, nodeLocations, numberOfLinks, waxmNet, location, maxNodeDistance, totalWaxProb, possibleLinks, waxmanLinksNormalized–PlacedonDart, waxmanLinkNormalizedCurrentIndex, totalNoOfPossibleLinks.

EdgeDensity is a double and as its name refers it stores the requested edge density. **Area** is an integer specifying the size of 2D plane. **NumOfNodes** is an integer which stores the number of nodes. WAXMANBeta and WAXMANAlfa are constants that used to compute Waxman probabilities. **NodeLocations** is a list that stores all node locations. **NumberOfLinks** is an integer storing how many links has been created. **WaxmNet** is a net object which is being built by Waxmanf. **Location** is a location object used to store temporary location information. **MaxNodeDistance** is a double corresponding to the maximum distance between two nodes in the network, this variable is used for probability

computation. **Possiblelinks** is a list that stores all possible links. **WaxmanLinksNormalizedAndPlacedonDart** is a tree map, it is used for lining up all the possible links according to their probabilities of existence. **WaxmanLinkNormalizedCurrentIndex** is a double which is used for keeping the current index of the tree map. **TotalNoOfPossibleLinks** is an integer which stores the total number of possible links.

Methods: Waxmanf has several methods. **AddWaxmanLink** method is used for adding a safe link to the network. Link selection is done according to the Waxman probability of existence. Any selected link is checked if it is already in the network or not. **CreatePossibleLinks** method creates all possible links between all possible vertex pairs. **CreateVertices** method creates the vertices, and places them randomly on 2D plane. **GetNextRandomSafeLink** method selects a link randomly among all possible links according to their possibility of existence and checks if the edge is already in the network or not. **NumberOfLinksInTheNetwork** method returns the current number of links in the network. **PutPossibleLinksToWaxmanDart** method takes an edge and puts it into line according to their probabilities of existence. For this purpose we use a tree map. We take all the possible links and put them into the tree map. When we want to add the next possible link to the tree map, we normalize its probability by dividing it by the total Waxman probability. After normalization, we put it in to the tree map at index waxmanLinkNormalizedCurrentIndex incremented by the normalized probability of the current link. Whenever we want to select a link, we select a random number and get the link corresponding to the ceiling key of that random number. **CreateLinks** method creates links using several of the previously described methods.

5.3.1.15 Caidaf

Usage: Caidaf stands for Caida network factory. The purpose of Caidaf is to take a data file prepared by CAIDA project and import the real Internet network topology into FRPcode. It takes the data file and creates a new node for each different AS and than create inter AS links according to the adjacency information in the file. Once the network topology is imported simulation experiments are done on the imported network.

Data Members: The data members of Caidaf are as follows. **NetSize** is an integer and it keeps the network size. **CurrW** is an integer and it defines the default weight for the new edges. **NumberOfLinks** is an integer and it keeps the current number of links. **NodeLocations** is an hasmap and it is used to store location of newly created vertex. **FromfileNet** is an instance of Net. It is the network that is being built. Whenever the network is completely built, the FromfileNet variable is returned to the calling function as the new network. **Location** is an instance of location which is used to store temporary location information. **InputFile** is an instance of file and it is used to reference to the input file that the Caida data is read from. **VerticesToCreate** is a string. In Caidaf, input data is a string and it is read in to VerticestoCreate variable. Then, further operations are continued using this string variable. **EdgesToCreate** is a string and it keeps the information about the edges to create. **TmpVertices** is a hashmap and it is used to store the temporary list of vertices while vertices are being built. Similarly, **tmpEdges** is a hashmap and it is used to store the temporary list of edges while new edges are being built.

Methods: CreateVertices method is used for creating the vertices from Caida data file. It reads the data file and creates a new vertex for each unique AS number in the file. Only one vertex is created for each AS number. **CreateLinks** method reads the Caida data file and creates links corresponding to adjacency information in the data file. **Create** method

calls previous two methods to create the Caida network. **NumberOfLinksInTheNetwork** method returns the current number of links in the network.

5.3.1.16 NetFromFilef

Usage: NetFromFilef stands for network from file factory. The purpose of NetFromFilef is to rebuilt the network from a text file. This text file is created by FRPcode as we request.

Data Members: The data members of NetFromFilef are as follows. **NetSize** is an integer and it keeps the network size. **CurrW** is an integer and it defines the default weight for the new edges. **NumberOfLinks** is an integer and it keeps the current number of links. **NodeLocations** is an hashmap and it is used to store location of newly created vertex. **FromfileNet** is an instance of Net. It is the network that is being built. Whenever the network is completely built, the FromfileNet variable is returned to the calling function as the new network. **Location** is an instance of location and it is used to store temporary location information. **InputFile** is an instance of file and it is used to reference to the input file that the network data is read from. **VerticesToCreate** is a string. In NetFromFilef, input data is a string and it is read in to VerticestoCreate variable. Then, further operations are continued using this string variable. **EdgesToCreate** is a string and it keeps the information about the edges to create.

Methods: CreateVertices method is used for creating the vertices from network data file. It reads the data file and creates a new vertex for each vertex in the file. **CreateLinks** method reads the network data file and creates links corresponding to edge information in the data file. **Create** method calls previous two methods to create the network. **NumberOfLinksInTheNetwork** method returns the current number of links in the network.

5.3.1.17 TestingNetf

Usage: TestingNetf stands for testing network factory. The purpose of TestingNetf is to create a grid-like network for testing purposes.

Data Members: The data members of TestingNetf are as follows. **NetSize** is an integer and it keeps the network size. **CurrW** is an integer and it defines the default weight for the new edges. **NumberOfLinks** is an integer and it keeps the current number of links. **NodeLocations** is an hashmap and it is used to store location of newly created vertex. **TestingNet** is an instance of Net. It is the network that is being built. Whenever the network is completely built, the TestingNet variable is returned to the calling function as the new network. **Location** is an instance of location and it is used to store temporary location information. **FactoryAgentSet** is a linked list and it is used to designate agent locations so that agent creation for testing net is defined implicitly.

Methods: CreateLinks creates grid like links between the nodes. **Create** method creates vertices, and calls previous method to create the edges and consequently, built the new network. **NumberOfLinksInTheNetwork** method returns the current number of links in the network.

5.3.1.18 Attackf

Usage: Attackf stands for attack factory. The purpose of **Attackf** is to provide the simulation with a unique interface for creating attacker set. **Attackf** has three children namely, **simpleAttack**, **incrementalAttackf** and **sequentialAttackf**. There is a static method called **makeAttackf** in Attackf. Whenever makeAttackf method is called with

network G and Attackf type as input, the attackf class creates a new attackf object and returns the link to the object. If requested attacker creation type is simple then a handle to simple attack factory is returned similarly if the requested type is sequential than a handle to sequential attack factory is returned and so on.

Data Members: Attackf defines three types namely, SIMPLE, INCREMENTAL and SEQUENTIAL.

Methods: Attackf has only one abstract method called create method. All child classes must have create method which creates an attacker set for the experiment and returns it to the requesting object.

5.3.1.19 SimpleAttackf

Usage: SimpleAttackf stands for simple attack factory. Its purpose is to create a set of attackers randomly and return it.

Data Members: It has two data members **net**. **Net** is an instance of Net class. **Na** is an integer which designates the number of attackers in the attacker set.

Methods: It has only one method named **create**. Create method selects *na* number of vertices randomly and places them in the set of attackers and returns the list of attackers.

5.3.1.20 SequentialAttackf

Usage: SequentialAttackf represents sequential attack factory. Its purpose is to select each vertex as an attacker one-by-one by sequentially returning the next vertex as the attacker.

Data Members: It has three data members; namely **net**, **remainingVertexSet**, and **tmpVertexSet**. Net is a handle to the current network, remainingVertexSet is a list of all the vertices, and tmpVertexSet is a list of vertices which are not selected to be victim yet.

Methods: It has only one method named **create**. Create method selects the next vertex from tmpVertexSet and returns it as the next attacker.

5.3.1.21 IncrementalAttackf

Usage: SequentialAttackf represents incremental attack factory. Its purpose is to select a set of attackers such that each attacker set is a subset of the next attacker set to be created.

Data Members: It has three data members; namely **net**, **remainingVertexSet**, and **tmpVertexSet**. Net is a handle to the current network, remainingVertexSet is a list of all the vertices, and tmpVertexSet is a list of vertices which are not selected to be attackers yet.

Methods: It has only one method named **create**. Create method selects takes the previous attacker set and add required number of new attackers to the set and returns the set as the new attacker set.

5.3.1.22 Agentf

Usage: Agentf stands for agent factory. The purpose of **Agentf** is to provide the simulation with a unique interface for creating agent set. **Agentf** has three children namely, **simpleAgentf**, **incrementalAgentf** and **GreedyAgentf**. There is a static method called

makeAgentf in Agentf. Whenever makeAgentf method is called with network G and Agentf type as input, the agentf class creates a new agentf object and returns the link to the object. If requested agent creation type is simple then a handle to simple agent factory is returned similarly if the requested type is sequential then a handle to sequential agent factory is returned and so on.

Data Members: Agentf defines three types, SIMPLE, INCREMENTAL and GREEDY.

Methods: Agentf has only one abstract method called create method. All child classes must have create method which creates an agent set for the experiment and returns it to the requesting object.

5.3.1.23 SimpleAgentf

Usage: SimpleAgentf stands for simple agent factory. Its purpose is to create a set of agents randomly and return it.

Data Members: It has three data members **net, agentDensity, na** . **Net** is an instance of Net class. **Na** is an integer which designates the number of agents in the agent set.**agentDensity** is a double which designates the density of agent to be created.

Methods: It has only one method named **create**. Create method selects *na* number of vertices randomly and places them in the set of agents and returns the list of agents.

5.3.1.24 IncrementalAgentf

Usage: IncrementalAgentf represents incremental agent factory. Its purpose is to move the agents such that the resulting values of M1 or M3 measures are optimized. IncrementalAgentf is implemented as a method in GreedyAgentf.

5.3.1.25 GreedyAgentf

Usage: GreedyAgentf represents greedy agent factory. Its purpose is to select edges greedily so that the agent placement optimizes either M1 value or M3 measurement values.

Data Members: It has several data members. **Net** is an instance of Net class and it is a reference to the current network G. **Na** is an integer which refers to the number of agents to be created. **Rt** is an instance of Routing Table(RT) class ant is it a reference to the current routing table R. **TmpRtf** is a reference to routing table factory (RTf). **AgentDensity** is a double and it designates the agent density. **TempLink M1Weights** is a hashmap which stores the edge versus benefit information for optimizing resulting M1 value. M1 benefit information for an edge used here is the number of paths passing through that edge. Benefit information is updated if there is currently no agent on the currently considered path. **TempLink M3Weights** is a hashmap which stores the edge versus benefit information for optimizing resulting M3 value. M3 benefit information for an edge used here is the calculated considering already placed agents and all possible placement of agents on the path. **SortedEdgeListM1** is a tree map which stores M1 benefit information in sorted order. Similarly, **SortedEdgeListM3** stores M3 benefit information in sorted order. Agents is an instance of agent class and it stores the newly created agents set.

Methods: Populate Temp Edge Weights M1 method populates the **Temp Link M1 Weights** hashmap according to the greedy M1 algorithm. **Update Edge Values M1**(Attacker,Victim) method updates the benefit information for the edges on the path from attacker to the victim. **Create Sorted Egde List For M1** method converts the TempLinkM1Weights into sortedEdgeListM1. **Get Next Agent M1** returns the next edge with maximum benefit.

Populate Temp Edge Weights M3 method populates the **Temp Link M3 Weights** hashmap according to the greedy M3 algorithm. **Update Edge Values M3** (Attacker,Victim) method updates the benefit information for the edges on the path from attacker to the victim. **Create Sorted Egde List For M3** method converts the TempLinkM1Weights into sortedEdgeListM3. **Get Next Agent M3** returns the next edge with maximum benefit. Increment method takes an agent set and replaces it so that resulting M1 and M3 values gets better.

5.3.1.26 RT

Usage: RT stands for routing table. Rt class keeps next hop information for all vertices to go to all the other vertices. It provides next hop service. Whenever it is given a source and destination vertex, it returns the next hop information.

Data Members: It has two data members. RtID is an integer which refers to the routing table id. RoutingTable is a hashmap. It stores next hop information for all the vertices in G.

Methods: GetNextHop(Source,Destination) method is used for requesting route information from the routing table. This method returns the next vertex to the requester.

5.3.1.27 RTf

Usage: RTf stands for routing table factory. The purpose of **RTf** is to provide the simulation with a unique interface for creating routing table. **RTf** has two children namely, **Dijkstra**, and **OnDemandDijkstra**. There is a static method called **makeRtf** in RTf. Whenever makeRTf method is called with network G and RTf type as input, the RTf class creates a new RTf object and returns the link to the requesting object. If the requested routing table creation type is Dijkstra then a handle to Dijkstra is returned similarly if the requested type is on demand Dijkstra then a handle to and demand Dijkstra is returned and so on.

Data Members: RTf defines two types namely, DIJKSTRA and ONDEMAND.

Methods: RTf has two abstract methods called create method and update method. All child classes must have create and update methods which creates and updates routing table respectively and returns it to the requesting object.

5.3.1.28 Dijkstra

Usage: Dijkstra stands for using Dijkstra's algorithm to create the routing table for G. It finds the path with lowest cost between a given source vertex (node) in the graph and every other vertex. The algorithm works as follows: It initially assigns to every node a distance value of infinity and source node a distance value of zero. Than it marks all nodes as unvisited. Than it sets source node as current and consider all unvisited neighbors of current node and calculate their distance from the initial node. If this distance is less than the previously recorded distance, it overwrite the previous distance. Once it is done

considering all neighbors of the current node we mark it as completed. A completed node will not be checked again. The distance recorded for a completed node is final and minimal. Then it sets the unvisited node with the smallest distance as the next "Current node" and repeat the procedures for all the nodes.

Data Members: **Net** is an instance of Net class and it is a reference to the current network G. **Rt** is an instance of Routing Table(RT) class ant is it a reference to the current routing table R. **RouteEntry** is a hash map and it stores routing table for one destination. **DijkstraRoutingTable** is a hash map which stores routing tables for all the nodes. **SortedDistToVertices** is a tree map and it is used to keep the current distance information of vertices, and get the next vertex to continue the algorithm from.

Methods: **InitializeRT** method assign to every node a distance value of infinity and source node a distance value of zero. **UpdateRT** runs Dijkstra'a algorithm for a single source. **PopulateRT** iteratively have the updateRt method run for all the vertices in order to create the whole routing table for G. **AddTosortedDistToVertices** method adds a given vertex to the SortedDistToVertices list. **GetNextFirstFromsortedDistToVertices** method removes and returns the first vertex from SortedDistToVertices list.

5.3.1.29 OnDemandDijkstra

Usage: OnDemandDijkstra stands for using Dijkstra's algorithm to create the routing table for a given destination only in G. It finds the path with lowest cost between the given source vertex (node) in the graph and every other vertex.

Data Members: **Net** is an instance of Net class and it is a reference to the current network G. **Rt** is an instance of Routing Table(RT) class ant is it a reference to the

current routing table R. **RouteEntry** is a hash map and it stores routing table for one destination. **DijkstraRoutingTable** is a hash map which stores routing tables for all the nodes. **SortedDistToVertices** is a tree map and it is used to keep the current distance information of vertices, and get the next vertex to continue the algorithm from.

Methods: InitializeRT method assign to every node a distance value of infinity and source node a distance value of zero. **UpdateRT** runs Dijkstra'a algorithm for a single source. **AddTosortedDistToVertices** method adds a given vertex to the SortedDistToVertices list. **GetNextFirstFromsortedDistToVertices** method removes and returns the first vertex from SortedDistToVertices list.

5.3.1.30 FRP

Usage: FRP is the class that refers to the FRP problem. Figure 5.3.2 shows the structure and relationships of FRPf. It keeps the necessary information for the FRP problem that is being solved.

Data Members: Net is an instance of Net class and it is a reference to the current network G. **Rt** is an instance of Routing Table(RT) class ant is it a reference to the current routing table R. **Agents** is an instance of agents class and it is a reference to the current set of agents. **Attack** is an instance of attack class and it is a reference to the current set attacks. **Victim** is an instance of victim class and it is a reference to the current victim. **AgentDensity** is a double and it stores the current agent density. **AttackNumber** is a double and it stores the current value of number of attackers.

Method: FRP does not have any method other than the constructor method.

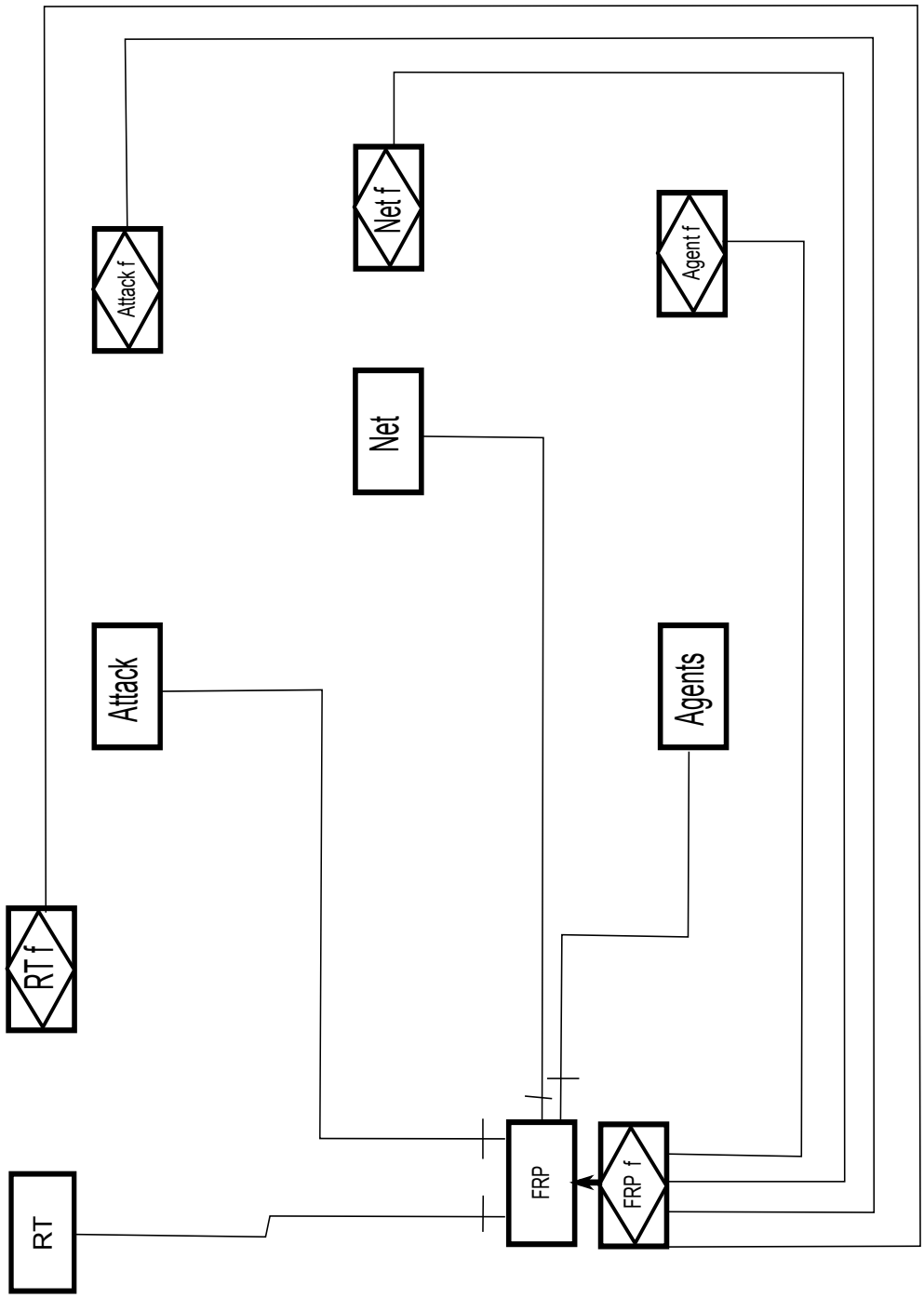


Figure 5.3.2: FRP code part two shows the structure and relationships of FRPf.

5.3.1.31 FRPf

Usage: FRPf stands for FRP factory. It is the class that is responsible for creating new instances of FRPs. It gets necessary objects from corresponding factories. In order to create a new **FRP**, it gets a new network from network factory, the routing table corresponding to the new network from routing table factory, the agent set from agent factory, the attacker sets from attacker factory and the victim from victim factory. Since FRP has 5 inputs, there are more than five different ways to create a new FRP. FRP can be created by using all new (G, R, A, D, v) or we can just change the victim or we can just change the attacker set or we can just change the agent set etc. Consequently, our FRPf must support these kind of operations. There is one thing to keep in mind that is if a new network is requested for the new FRP, then we must create routing table, agent set and attacker set and victim for the new network again.

Data Members: Net is an instance of Net class and it is a reference to the current network G. **Rt** is an instance of Routing Table(RT) class and it is a reference to the current routing table R. **Agents** is an instance of agents class and it is a reference to the current set of agents. **Attack** is an instance of attack class and it is a reference to the current set attacks. **Victim** is an instance of victim class and it is a reference to the current victim. **AgentDensity** is a double and it stores the current agent density. **AttackNumber** is a double and it stores the current value of number of attackers. **FRP** is an instance of FRP and it is used to store the current FRP.

Netf is an instance of netf class and it is a reference to the current network factory. **Rtf** is an instance of Rtf class and it is a reference to the routing table factory. **Rtf.Type** is an instance of Rtf.Type class and it is a reference to the current routing table factory type. **Agentf** is an instance of agents class and it is a reference to the current agent factory.

Agentf.Type is an instance of Agentf.Type class and it is a reference to the current agent factory type. **Attackf** is an instance of attack factory class and it is a reference to the current attack factory. **Attackf.Type** is an instance of Attackf.Type class and it is a reference to the current attack factory type. **Victimf** is an instance of victim factory class and it is a reference to the current victim factory. **Victimf.Type** is an instance of Victimf.Type class and it is a reference to the current victim factory type.

Methods: There are several method that the FRPf supports. **Advance**(boolean newRT, boolean newAgent, boolean newAttack, boolean newVictim) method creates new objects. If **newAgent** parameter is true, than this method creates a new set of agents. If **newAttack** parameter is true, than this method creates a new set of attackers. **newVictim** parameter is true, than this method creates a new victim. **newRT** parameter is true, than this method creates a new routing table. **Advance**(boolean newAgent, double newAgentDensity, boolean newAttack, int newAttackNumber). If **newAgent** parameter is true, than this method sets the current agent density to the new agent density and creates a new set of agents. If **newAttack** parameter is true, than this method sets the current number of attackers to the new number of attackers and creates a new set of attackers. **IncrementAgentSetGreedy** method is used for incremental agent placement. Whenever this method is called the existing agent set is incremented to achieve a better M1 or M3 result. **NewNetwork** method crates a new network. After creating the new network all of the factories which take network as an input are updated with the new network information. **Create** method creates a new FRP using the most up to date G, R, A, D, v

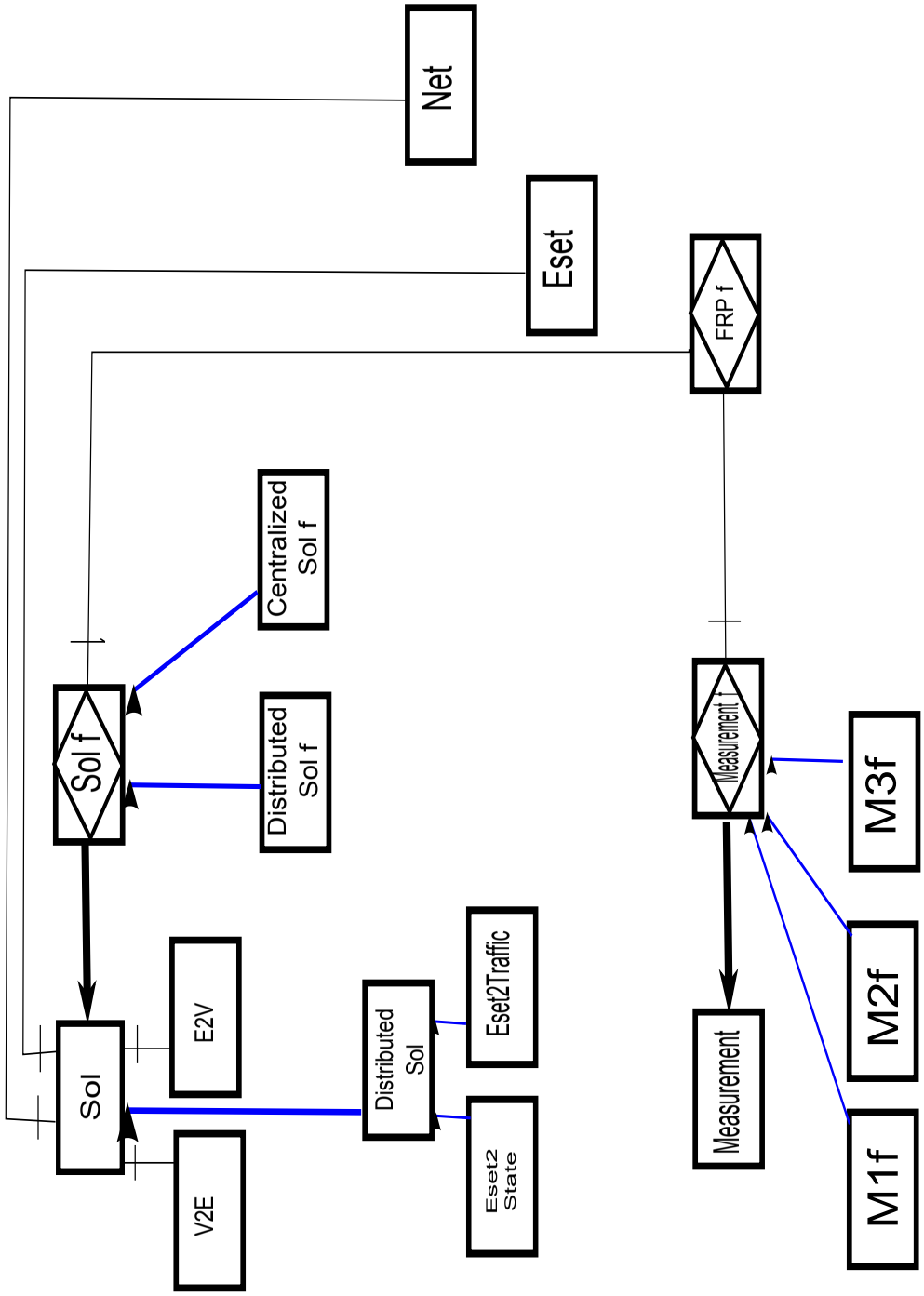


Figure 5.3.3: FRP code part three shows the structure and relationships of Solf and Measuremenf.

5.3.2 Code description

5.3.2.1 Sol

Usage: **Sol** refers to solution of FRP problem. Figure 5.3.3 shows the structure and relationships of Sol. It stores the necessary information about the solution of FRP problem that is being solved. It provides measurement class with useful information to make the measurements. It stores information about the agents and logical links between the agent in the solution set. It also provides information about the attackers distance to the first agent on their paths to the victim and to the victim.

Data Members: SolAgents is a hash map. It keeps records of the agents in the solution set. Similarly, **solLogicalLinks** is a hash map that keeps the records about the logical links between the agents that are in the solution set. **SolAttackersAgents** is also a hash map. For each attacker, the SolAttackersAgents keeps the list of agents that are on attackers path to the victim. **SolAttackersDistToFirstAgents** is another hash map. It stores each attacker's distance to the first agent on their path to the victim. Lastly, the **SolAttackersDistToVictim** stores each attacker's distance to the victim.

Methods: GetSolutionAgents method returns the list of agent that are in the solution set. **GetAttackersAgents** method returns the mapping from attackers to their agents. **GetAttackersDistToFirstAgents** method returns the mapping from attackers to their closest agent on the path to the victim. Finally, the **getAttackersDistToVictim** method returns the mapping from attackers to their distances to the victim.

5.3.2.2 Solf

Usage: **Solf** stands for solution factory. The purpose of Solf is to provide the simulation with a unique interface for creating solutions. Solf has two children namely, **CentSolf** and **DistSolf**. There is a static method called **makeSolf** in Solf. Whenever makeSolf method is called with the problem FRP and requested solution type as input, the solf class creates the solution factory object and returns the link to the requesting object. If the requested solution creation type is *central* then a handle to central solution factory is returned similarly if the requested type is distributed then a handle to distributed solution factory is returned and so on.

Data Members: Solf defines two types namely, CENTRAL and DISTRIBUTED.

Methods: Solf has only one abstract method called create method. All child classes must have create method which creates the solution to the FRP problem for the current experiment and returns it to the requesting object.

5.3.2.3 CentSolf

Usage: CentSolf stands for centralized solution factory. Its purpose is to create the solution to the current FRP problem and return the solution. It takes G, R, D, A, v as input and returns the list of agents and the logical links to the requesting object. Besides, it provides some additional data which will ease the measurements.

Data Members: SolAgents is a hash map. It keeps records of the agents in the solution set. Similarly, **solLogicalLinks** is a hash map that keeps the records about the logical links between the agents that are in the solution set. **SolAttackersAgents** is also a

hash map. For each attacker, the SolAttackersAgents keeps the list of agents that are on attackers path to the victim. **SolAttackersDistToFirstAgents** is another hash map. It stores each attacker's distance to the first agent on their path to the victim. Lastly, the **SolAttackersDistToVictim** stores each attacker's distance to the victim. **FRP** is an instance of the FRP problem. **Net** is an instance of Net class and it is a reference to the current network G. **Rt** is an instance of Routing Table(RT) class and it is a reference to the current routing table R. **Agents** is an instance of agents class and it is a reference to the current set of agents. **Attack** is an instance of attack class and it is a reference to the current set attacks. **Victim** is an instance of victim class and it is a reference to the current victim.

Methods: GetAllAgentsOnPathToSolution(Attacker, victim) method takes an attacker and a victim as input. Starting from the attacker, this method visits each link on attackers path to the victim and checks if there is an agent on the link or not. If there is an agent on the link then the information about the agent is written to the list called *SolAttackersAgents*. Besides this, if an agent is the closest agent to the attacker, then the distance of the attacker to this agent is also recorded in another mapping called *SolAttackersDistToFirstAgents*. When the victim is reached, the attacker distance to the victim is recorded in *SolAttackersDistToVictim*.

5.3.2.4 Measurement

Usage: Measurement refers to one of the different measures of FRP solution. It is actually an object which holds the result of one of the M1, M3, M3, M4, and convergence measures.

Data Members: It has following data members. **NetId** is an integer and it refers to

the network id for which this measurements are done for. **NumOfNodes** is an integer and it stores the number of nodes in the network. **DensityOfAgents** is a double and it stores the agent density corresponding to this measurement. **NumOfAttackers** is a double and it stores the number of attackers. **NumOfLinks** is an integer and it shows the number of links in the network. **Victim** is an instance of victim class and it refers to the current victim. **Result** is a double and it stores the result of the measurement. Its value depends on the measurement factory that has created this measurement. For example, if the measurement is created by M1f, M1 measurement factory, then the result corresponds to M1 value of the current solution. **MeasurementType** is a string and it show the type of the measurement (i.e. M1,M2...).

Methods: Measurement has one method which is **writeToFile**. The *writeToFile* function writes the results to an external file.

5.3.2.5 Measurementf

Measurementf stands for measurement factory. The purpose of **Measurementf** is to provide the simulation with a unique interface for creating the measurements. **Measurementf** has four children namely, **M1f**, **M2f**, **M3f**, **M41f**, and **Convf**. There is a static method called **makeMeasurementf** in Measurementf. Whenever measurementf method is called with FRP and *solution* as input, the measurementf class creates a new measurementf object and returns the link to the requesting object. If requested measuremtf creation type is M1f then a handle to M1f is returned similarly if the requested type is M2f then a handle to M2f is returned and so on.

Data Members: Measurementf defines five types namely, M1f, M2f, M3f, and convf.

INCREMENTAL and GREEDY.

Methods: Measurementf has two abstract methods called **create** method and **getResult** method. All child classes must have both *create* and *getResult* method one of which creates the measurement set and the other one returns the results of the experiment.

5.3.2.6 M1f

Usage: M1f stands for measurement M1 factory. M1f is responsible from getting an instance of APRP and solution to the FRP and producing the M1 result.

Data Members: It has three data members. FRP is an instance of FRP an it refers to the FRP under consideration. Sol is an instance of solution and it refers to the solution to the current victim. result is a double and it stores the result of the measurement.

Methods: M1f has three methods. **CreateMeasuremet** method is the main method that calculates the M1f value. **CreateMeasuremet** method creates the measurement. **GetResult** method returns the result.

5.3.2.7 M2f

Usage: M2f stands for measurement M2 factory. M2f is responsible from getting an instance of APRP and solution to the FRP and producing the M2 result.

Data Members: It has three data members. FRP is an instance of FRP an it refers to the FRP under consideration. Sol is an instance of solution and it refers to the solution to the current victim. result is a double and it stores the result of the measurement.

Methods: M2f has three methods. **CreateMeasuremet** method is the main method that calculates the M2f value. **CreateMeasuremet** method creates the measurement. **GetResult** method returns the result.

5.3.2.8 M3f

Usage: M3f stands for measurement M3 factory. M1f is responsible from getting an instance of APRP and solution to the FRP and producing the M3 result.

Data Members: It has three data members. FRP is an instance of FRP an it refers to the FRP under consideration. Sol is an instance of solution and it refers to the solution to the current victim. result is a double and it stores the result of the measurement.

Methods: M3f has three methods. **CreateMeasuremet** method is the main method that calculates the M3f value. **CreateMeasuremet** method creates the measurement. **GetResult** method returns the result.

5.3.2.9 Convf

Convf stands for measurement convergence type factory factory. Convf is responsible from getting an instance of APRP and solution to the FRP and producing the transient result.

Data Members: It has three data members. FRP is an instance of FRP an it refers to the FRP under consideration. Sol is an instance of solution and it refers to the solution to the current victim. result is a double and it stores the result of the measurement. **Methods:** Convf has three methods. **CreateMeasuremet** method is the main method

that calculates the Convf value. **CreateMeasuremet** method creates the measurement. **GetResult** method returns the result.

5.3.2.10 DistSolf

Usage: **DistSolf** stands for distributed solution factory. Its purpose is to create the solution to the current FRP problem such that we can get transient measures for the FRP problem. It takes FRP as input and returns the transient measures data, list of agents and the logical links to the requesting object. Besides, it provides some additional data which will ease the measurements. In order to measure the transient measures, the DistSolf first have a shadow copy of the FRP problem created in distributed simulation environment and then produce the solution.

Data Members: NB is an instance of netBuilder. **TmpProgressReport** is an instance of progress report and it is used for keeping records about the state changes of the objects in distributed simulation. **DAS** is an instance of Distributed FRP simulator. **SolAgents** is a hash map. It keeps records of the agents in the solution set. Similarly, **solLogicalLinks** is a hash map that keeps the records about the logical links between the agents that are in the solution set. **SolAttackersAgents** is also a hash map. For each attacker, the SolAttackersAgents keeps the list of agents that are on attackers path to the victim. **SolAttackersDistToFirstAgents** is another hash map. It stores each attacker's distance to the first agent on their path to the victim. Lastly, the **SolAttackersDistToVictim** stores each attacker's distance to the victim. **FRP** is an instance of the FRP problem. **Net** is an instance of Net class and it is a reference to the current network G. **Rt** is an instance of Routing Table(RT) class and it is a reference to the current routing table R. **Agents** is an instance of agents class and it is a reference to the current set of agents. **Attack** is

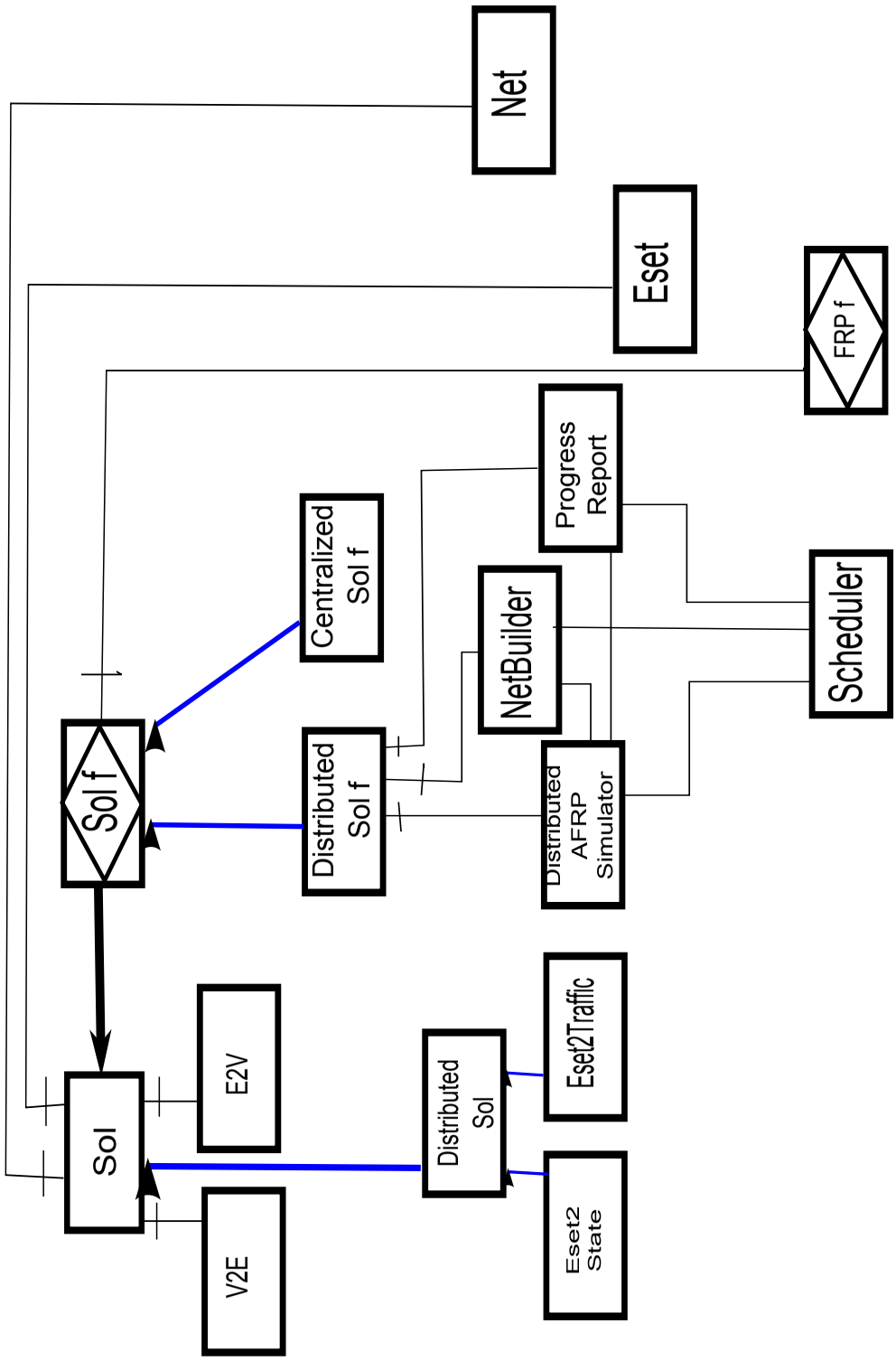


Figure 5.3.4: FRP code part four shows the structure and relationships of DistSolf.

an instance of attack class and it is a reference to the current set attacks. **Victim** is an instance of victim class and it is a reference to the current victim.

Methods: Create method is used to create the solution. Once the create method is called, a shadow copy of the FRP problem is created in distributed simulation environment, distributed event simulator is started and transient measures are measured. Upon completion of the distributed event simulation, the results are created and solution is produced. **GetAllAgentsOnPathToSolution(Attacker, victim)** method takes an attacker and a victim as input. Starting from the attacker, this method visits each link on attackers path to the victim and checks if there is an agent on the link or not. If there is an agent on the link then the information about the agent is written to the list called *SolAttackersAgents*. Besides this, if an agent is the closest agent to the attacker, then the distance of the attacker to this agent is also recorded in another mapping called *SolAttackersDistToFirstAgents*. When the victim is reached, the attacker distance to the victim is recorded in *SolAttackersDistToVictim*.

5.3.2.11 ProgressReport

Usage: ProgressReport stands for progress report. Its purpose is to record important events for a distributed event simulation. It stores individual reports reported by simulation entities. It also stores the time of last event recorded, the start time of the attack, the stop time of the attack , the start time of last experiment, the total time elapsed until system become stabilized after the attack start and after the attack stopped.

Data Members: Report is a hash map that stores all incoming progress reports. **Last Event Added** is a double and it stores the time when the last event is added. **Time**

Attack Started is a double and it stores the time when the attack has started. **Time Attack Stopped** is a double and it stores the time when the attack has ended. **Time Last Experiment Started** is a double and it stores the time when the current experiment has started. **Time Agents States Got Stable During Attack** is a double and it stores how long did it take for the agent system to become stable after the attack has started. **Time Agents States Got Stable After Attack** is a double and it stores how long did it take for the agent system to become stable after the attack has ended.

Methods: AddEntry method adds an entry to the progress report repository. **Get Last Event Added** method returns the value of **Last Event Added**. Similarly, **get Time Agents States Got Stable After Attack**, **get Time Agents States Got Stable During Attack**, **get Time Attack Started**, **get Time Attack Stopped**, **get Time Last Experiment Started**, returns the values of corresponding variables.

5.3.2.12 NetBuilder

Usage: NetBuilder stands for network builder. Its purpose is to create the necessary simulation entities such that those simulation entities can communicate over discrete event simulation system. Other than the regular NetBuilder components, the most important part of NetBuilder in our experiments is its creating the copy of *FRP* problem in discrete event environment. NetBuilder reads from FRP and creates all the vertices corresponding to the vertices in *FRP* and creates all the edges as specified in *FRP*.

Data Members:

Methods: FRPDistribued(FRP)method reads from FRP and creates simulation entities for each vertices, and agents. It also creates the copy of FRP problem in discrete event

environment. **StartAttack** method starts the attack simulation, and **stopattack** method stops the ongoing attack.

5.3.2.13 DistributedFRPSimulator

Usage: DistributedFRPSimulator stands for distributed FRP simulator. The purpose of DistributedFRPSimulator is to control the simulation. Whenever an instance of DistributedFRPSimulator is created it is given the reference of current network builder and the progress report. It controls when the simulation starts and when it finishes. Once the simulation is run, it adds an entry to the progress report object and tells the network builder to start attack simulation. It also schedules a task for itself for a future time; at each time it receives the task alert from scheduler it checks if there is any change in the progress report object. If there is no change in the progress report then the simulation is stopped.

Data Members: The DistributedFRPSimulator(DAS) has two data members. NB is an instance of network builder and it references to the network builder object of the current simulation. PR is an instance of progress report. It is a repository that stores all the events reported by simulation entities.

Methods: Run method runs the experiment by asking network builder to start the network simulation and DDoS attack. **Recv** method deals with incoming event messages; every time a message is received, DAS controls the progress report for changes. If there is no change then DAS stops the experiment.

CHAPTER 6

SYSTEM EVALUATION

6.1 Steady state measures

6.1.1 Influence of agent density on performance

Purpose. We seek to investigate the influence of agent density[1] on performance measures M1, M2 and M3 for networks of different sizes. We will repeat the experiments for Waxman networks of sizes: 50 nodes, 100 nodes, 200 nodes, 500 nodes, 1000, and 2000 nodes.

Experiment setup. We have implemented our system as a distributed protocol. We then implemented a simulation of our distributed system based on these protocols. The simulation is then used at the heart of the following procedure:

1. Set the network size $|V| = n$ and create Waxman network G of this size. This is done by taking a geometric square of side $5\sqrt{|V|}$. Within this we place $|V|$ nodes uniformly at random. This establishes the vertex set V. Then we add random links between nodes, chosen with probability that is inversely proportional (exponentially) to the Euclidean distance of two nodes. This link addition process continues until the desired edge density is reached. Once the desired edge density is reached, we check

1. By agent density we mean the ratio of the number of agents to the number of links.

if the network is connected or not. If it is we return the network. Otherwise, we continue to add edges until the graph becomes connected. In our experiments, we took the edge density to be slightly above two (2.1); this value was chosen because it is close to the value obtained through analysis of the present Internet topology [68]. This establishes the edge set E. In this way, we have constructed a graph $G = (V, E)$ which has the form of a Waxman network whose edge density reflects present edge densities in the Internet[2].

2. Instantiate a routing table R for this network by running Dijkstra's algorithm from all nodes.

3. Set the agent density ϵ.

4. Set attacker density δ.

5. For each of (100) agent sets of size ϵm, which are uniformly randomly selected.

6. For each of (100) attacker sets of size δn, which are uniformly randomly selected.

7. Consider each node v in V, in turn, as a victim.

8. Create an instance flow reconstruction problem (FRP) (G, R, A, D, v).

9. Obtain a maximal valid solution and compute M1, M2, and M3 for the solution.

Z. Repeat the experiment for new value of agent density ϵ; keep the values of the rest of the variables the same.

---

2. In the implementation, the network is created by Waxman network factory as described in section 5.3.1.14, agents are created and placed on the links randomly by agent factory as described in section 5.3.1.22, attackers are selected randomly by attacker factory as described in section 5.3.1.18, and victims are selected by victim factory as described in section 5.3.1.10.

Results. Figure 6.1.1 shows that as the agent density increases from 1% to 15%, the percentage of attackers that remain undiscovered drops from 90% to under 50%. As the agent deployment grows further to 28%, only 20% of the attackers (on average) are able to evade detection. More importantly we see that for any given deployment level, the

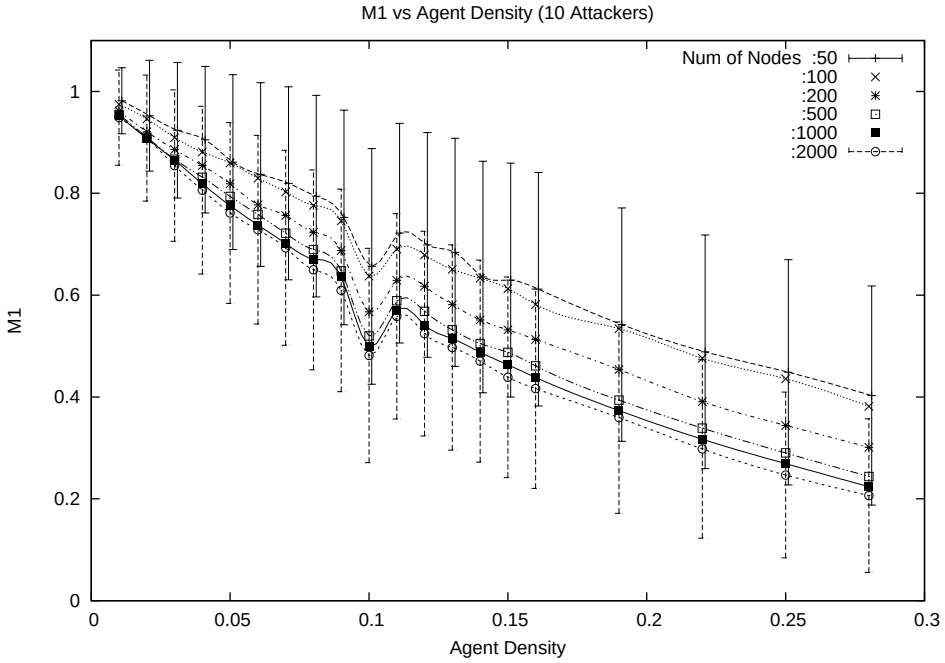


Figure 6.1.1: M1 versus agent density

performance improves as the networks under consideration become larger. For example, our experiments showed that with a 15% deployment of agents in a network of 50 nodes, almost 62% of the attackers remained undiscovered–but when the network under consideration grew to 2000 nodes, the same 15% deployment yielded much better performance–only 42% of the attackers evade detection. This analysis indicates that the proposed solution scales, performing increasingly better in larger networks. The error bars in the graph, while large, are also seen to shrink as the network size increases. For example, at 25% deployment, the M1 measure of a 50 node network has a standard deviation of 20%, while for 2000 node networks, the variance shrinks to 14%. The high variances observed are to be expected, since the performance measures are sensitive to the choice of attackers, agents and victim–

all of which are random. The lower boundary of the error bars indicate that some agent set placements perform much better than others; describing the criteria under when this occurs is the subject of ongoing research by the authors.

Figure 6.1.2 examines the dependence of measure M2 on agent density in the network. Recall that M2 is distance between the furthest agent in the reconstructed flow and the nearest corresponding attacker. A low mean value of M2 indicates that the system was

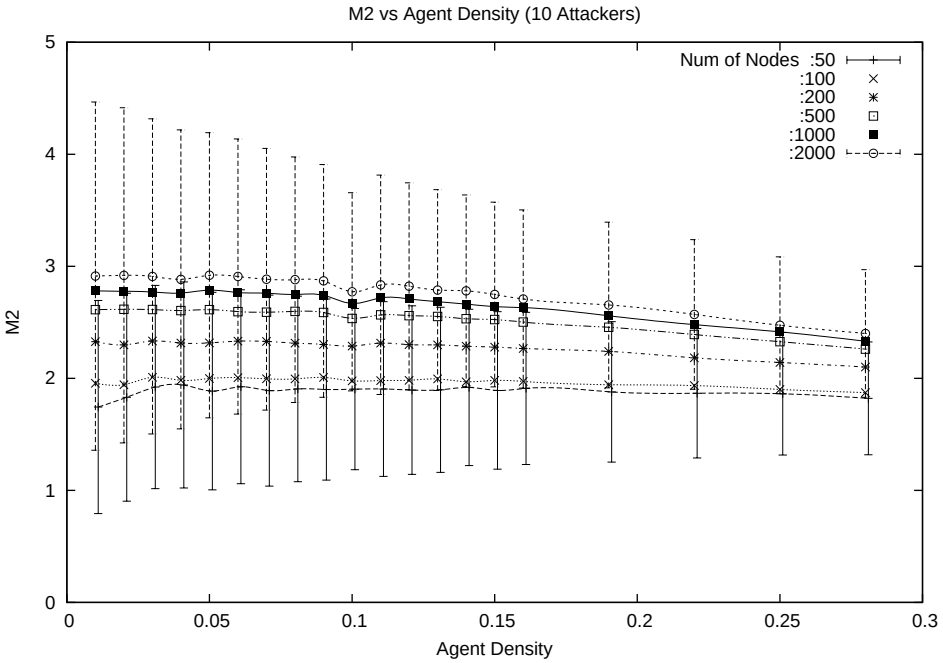


Figure 6.1.2: M2 versus agent density

able to trace back closer to the attacker. As the figure shows, in our experiments we were able to trace back to (on average) 1.8 hops from attacking nodes in networks of 50 nodes, and to within 2.7 hops from attackers networks of 2000 nodes. Although the system was not able to get as close to attackers in larger networks (e.g. 2.7 hops versus 1.8 hops) the rate at which system performance degraded was insignificant compared to the increases in network size and diameter. Considering that the agents were placed randomly, it is quite remarkable that we are able to trace back the attack flows to within 1.8 hops of an

attacker[3]. Having such traceback information that reaches within 1-2 hops of an attack sources could be sufficient to localize it completely, if the same compromised machine (e.g. bot within a botnet) takes part in several attacks over its lifetime. In this case, a hop-wise triangulation strategy could be used to pinpoint the true attack source.

Figure 6.1.3 depicts the behavior of M3 with respect to agent density. Recall that M3 is the normalized distance between the furthest agent in the reconstructed flow and the nearest corresponding attacker: An M3 value of 0.0 signifies that traceback was able to reconstruct the flow to the precise autonomous system in which the attacker resides. A value of 0.5

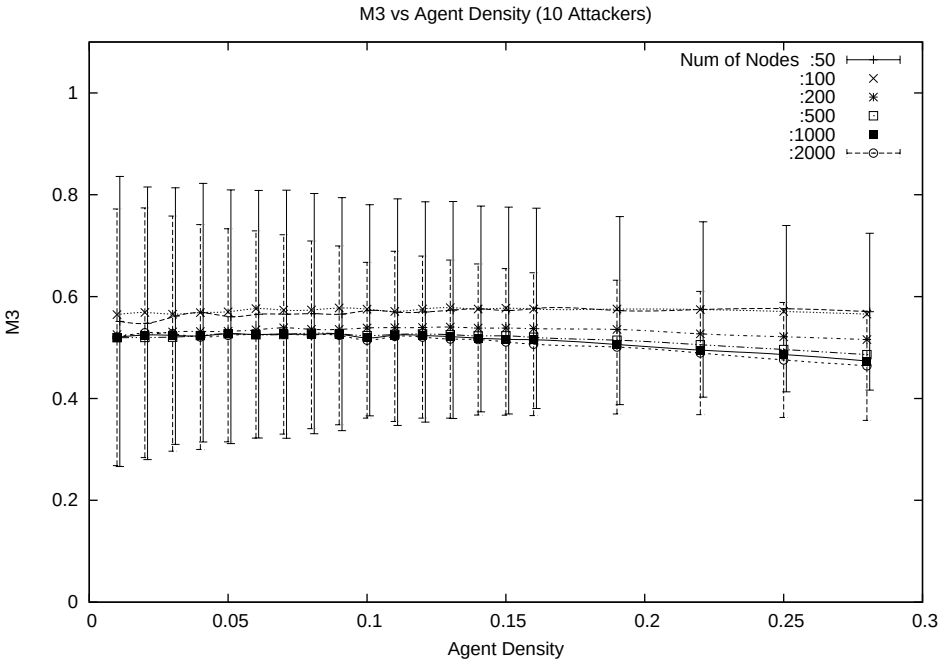


Figure 6.1.3: M3 versus agent density

means flow reconstruction was only possible (on average) halfway from the victim to the attacker. The graph shows that, like M2, the normalized distance to the attacker does not change significantly as deployment increases. More precisely, while M3 decreases slightly as

3. A further technical refinement can be obtained by noting that we know the incoming interface of attack flow at every agent, and this allows us to lower the hop count values further by one hop.

the network size increases from 50 to 500, the extent of the decrease stabilizes for networks much larger than 500. The error bars in the figure show that the variance in M3 decreases as agent density increases, since system performance is less susceptible to the "Bad" placement of a few agents. On the whole, the graph shows us that in large networks, even a modest deployment of our system will allow agents to reconstruct the attack flow halfway up to the attacking node.

Figure 6.1.4 depicts the same result as the figure 6.1.1. However, in this case we wanted to clearly see how do the standard deviation curves (STD) behave for the M1 value. We saw

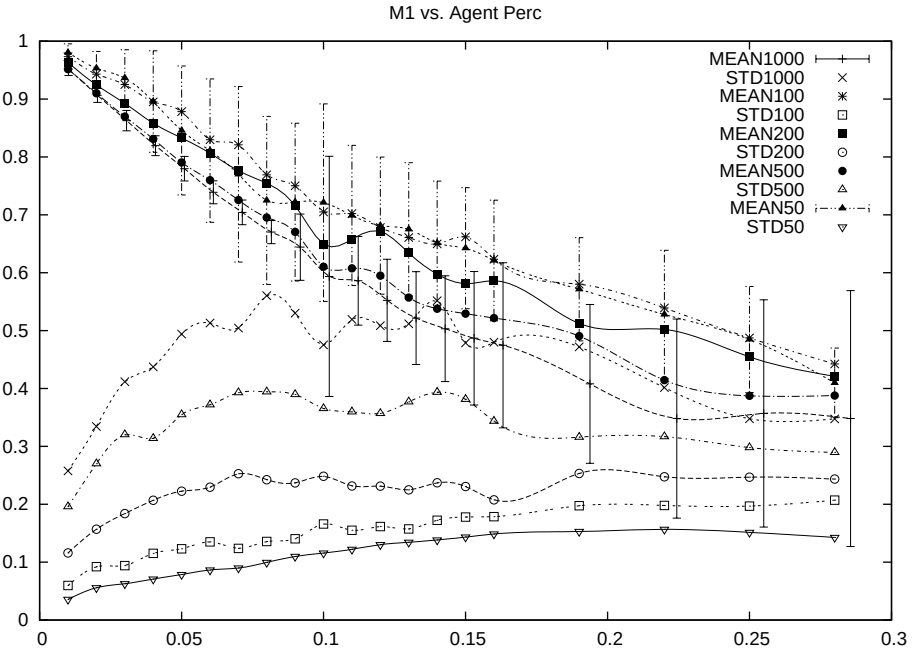


Figure 6.1.4: M1 versus agent density with STD

that the STD curves increases starting from %0.01 agent density until agent density reaches %0.1. It stays there until agent density reaches %0.15 and drops again. STD values seems to be high which show that there are some very good results and some very bad results. Since high STD values show that there are some very good results, they encouraged us on focusing on tuning our system such that the results are always very good. We will talk

about the improved version of the system later in this document.

6.1.2 Influence of attacker density on performance

Purpose. The previous experiment considered a fixed number (10) of attacking nodes. What if there are more attackers? Purpose of this experiment is to investigate the influence of number of attackers on performance measures M1, M2 and M3 for networks of different sizes. We will do experiments to see the results when this assumption is lifted. Specifically, we will allow the number of attackers to vary as a *percentage* of the network size. We will consider attacks which are coordinated by different sized sets of attackers (e.g. botnets of different sizes). The size of the attacker set will vary between between 1% and 10% of the network size. Throughout this experiment, we will keep the agent deployment density fixed at 25%. We will repeat the experiments for the networks of sizes: 50 nodes, 100 nodes, 200 nodes, 500 nodes, and 1000.

Experiment setup. The setup is as described for the previous experiment (Section 6.1.1), with the one change to step:

Z. Repeat the experiment for new value of attacker density δ; keep the values of the rest of the variables the same.

Results. Figure 6.1.5 shows the relation between attacker density and the value of M1. There are five curves in the graph representing different network sizes 50, 10, 200, 500, and 1000 respectively. For all the curves, the M1 value shows a flat pattern as the attacker density increases. However, as the network size increases the value of M1 decreases, which means we detect more attackers. Curves representing 50 nodes and 100 nodes crosses each-

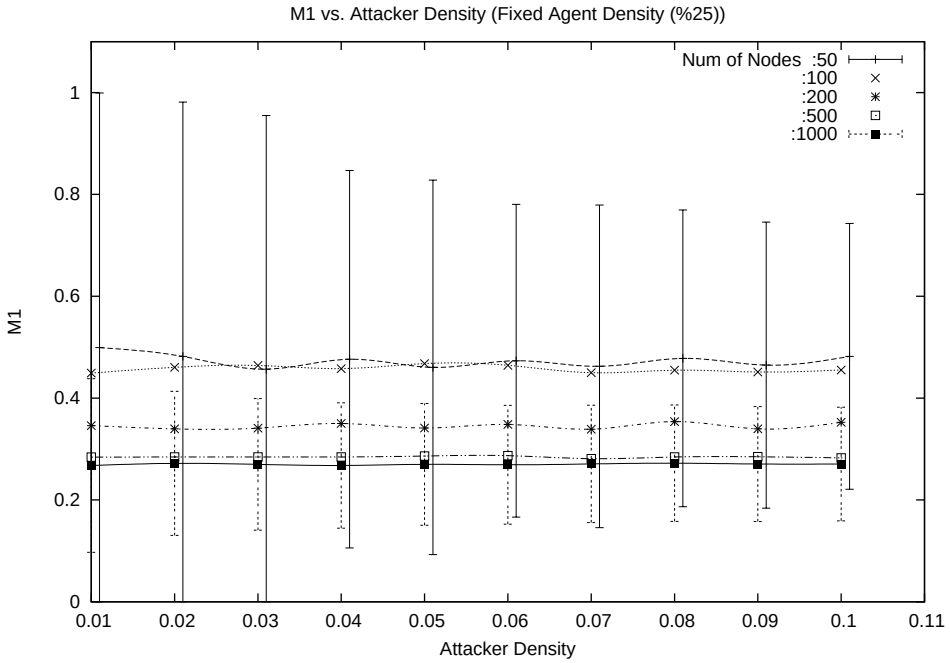


Figure 6.1.5: M1 versus attacker density

other at different points; however, they keep their almost flat behavior. The rest of the curves does not cross each other; moreover, as the net size increases curves become flatter. The size of the error bars decreases as the network size increases. As a result, we conclude that the attacker density does not effect the value of M1. Accordingly, in the experiments that follow we will restrict our attention to scenarios in which the set of attackers has a fixed size of 10.

Figure 6.1.6 shows the relation between attacker density and the value of M2. There are five curves in the graph representing different network sizes 50, 10, 200, 500, and 1000 respectively. For all the curves, the M2 value shows a flat pattern as the attacker density increases. However, as the network size increases the value of M2 increases, which means we go further from the attacker. As a result, we conclude that the attacker density does not effect the value of M2. Accordingly, in the experiments that follow we will restrict our attention to scenarios in which the set of attackers has a fixed size of 10.

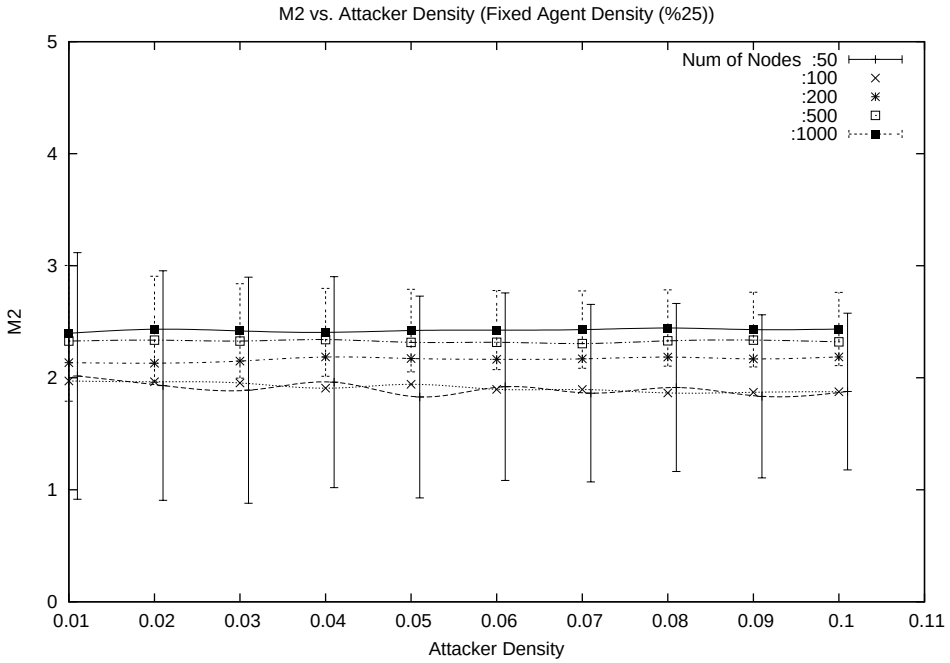


Figure 6.1.6: M2 versus attacker density

Figure 6.1.7 shows the relation between attacker density and the value of M3. There are five curves in the graph representing different network sizes 50, 10, 200, 500, and 1000 respectively. For all the curves, the M3 value shows a flat pattern as the attacker density increases. However, as the network size increases the value of M3 decreases. Recall that M3 is the attackers normalized distance to the first agent, which means when considered the distance to the victim we get closer the attacker. As a result, we conclude that the attacker density does not effect the value of M3. Accordingly, in the experiments that follow we will restrict our attention to scenarios in which the set of attackers has a fixed size of 10.

Graphs 6.1.5, 6.1.6, and 6.1.7 clearly show that for a fixed agent density and fixed network size, the performance values M1, M2, and M3 does not vary significantly in proportion to the number of attacking nodes. Accordingly, in the experiments that follow we will restrict our attention to scenarios in which the set of attackers has a fixed size of 10.

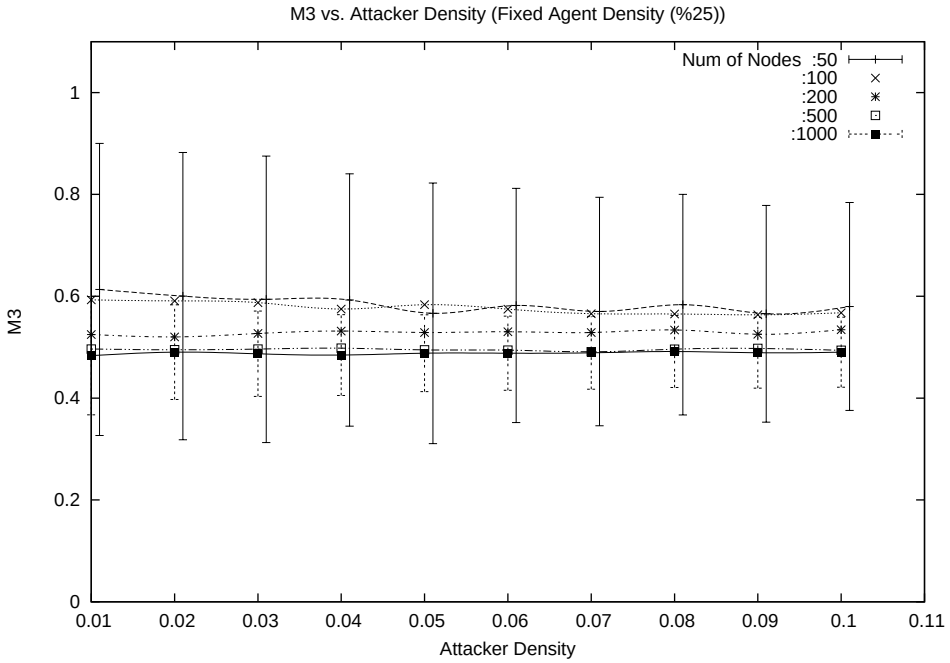


Figure 6.1.7: M3 versus attacker density

6.1.3 Experiment sensitivity to the number of agent sets

Purpose. The purpose of this experiment is to investigate how will the performance measures be effected by the number of agent sets used for the experiment. Recall that we used 100 different agent sets for the experiments 6.1.1 and 6.1.2. Upon completion of this experiment, we want to find out if the results of experiments 6.1.1 and 6.1.2 are valid in general or not. in another word, would the results be relevant if we had used a greater or smaller number of different agent sets.

Experiment setup. The experiment operates as follows:

- Create a Waxman network $G = (V, E)$ of size 500 and let R be the routing table on G derived by Bellman-Ford's algorithm.
- Set the agent density ϵ to 15%

- Set the number of attackers $|D|$=10.
- For each of $\mathbf{X}$ agent sets of size $\epsilon|E|$, which are uniformly randomly selected.
- For each of (100) attacker sets of size $\delta|V|$, which are uniformly randomly selected.
- Consider each node v in V, in turn, as the victim.
- Create an instance FRP problem (G, R, A, D, v).
- Obtain a maximal valid solution and compute M1, M2, and M3 for the solution.

Z. Repeat the experiment for the $\mathbf{X}$=5, 10, 20, 50, 100, while keeping the values of the rest of the variables the same.

Results. Figure 6.1.8 presents the dependence of M1 value on the number of agent sets. The curve shows the mean value of M1 for different experiments with different number of agent sets. Although not perfect, the M1 curve shows constant relationship between M1 performance as the number of agent sets increases. The error bars shows the variances of the M1 values for each experiment. The variance between the size of error bars gets smaller as we move from x=5 to x=10, and it stays almost same for x=20, then it increases and then decreases again for x=100. Although, there is variance between the sizes of error bars, the change is neither monotonic nor significant. We conclude that the number of agent sets does not influence the value or variance of M1 significantly.

Figure 6.1.9 presents the dependence of M2 value on the number of agent sets. The curve shows the mean value of M2 for different experiments with different number of agent sets. Similar to the M1, the M2 curve shows a flat behavior as the number of agent sets increases. The error bars shows the variances of the M2 values for each experiment. The error bars are almost the same in size. Consequently, we say that the number of agent sets does not influence the value of M2 significantly.

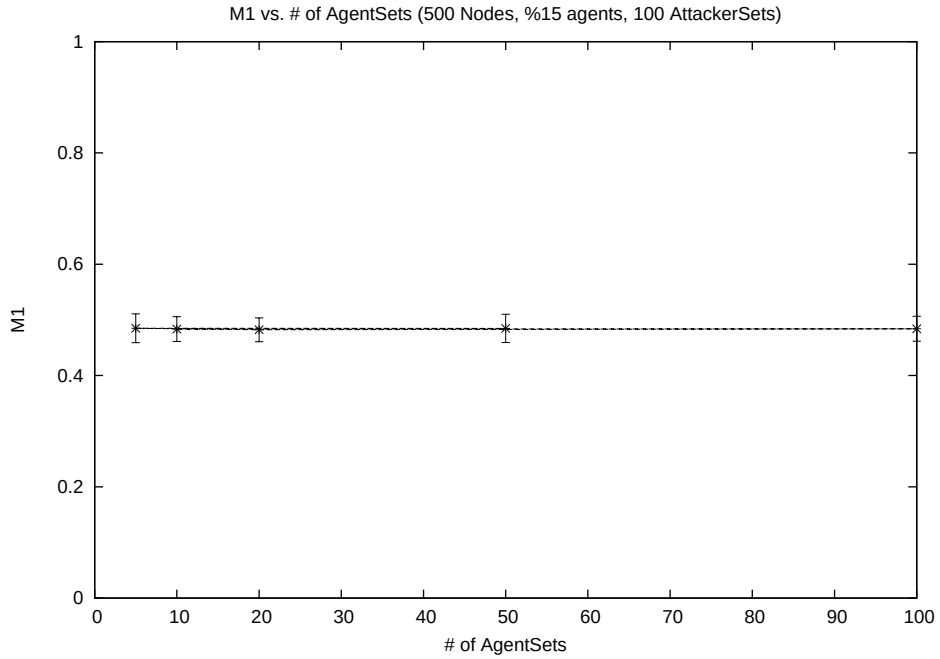


Figure 6.1.8: M1 versus number of agent sets

Figure 6.1.10 presents the dependence of M3 value on the number of agent sets. The curve shows the mean value of M3 for different experiments with different number of agent sets. Similar to the M1 and M2, the M3 curve shows a flat behavior as the number of agent sets increases. The error bars shows the variances of the M2 values for each experiment. The error bars shows small variances, however, it is neither monotonic nor significant. Consequently, we say that the number of agent sets does not influence the value of M3 significantly.

Figures 6.1.8, 6.1.9, 6.1.10 clearly show that the performance values M1, M2, and M3 does not vary significantly in proportion to the number of agent sets. Consequently, the results of experiments 6.1.1 and 6.1.2 are valid in general for different number of agent sets.

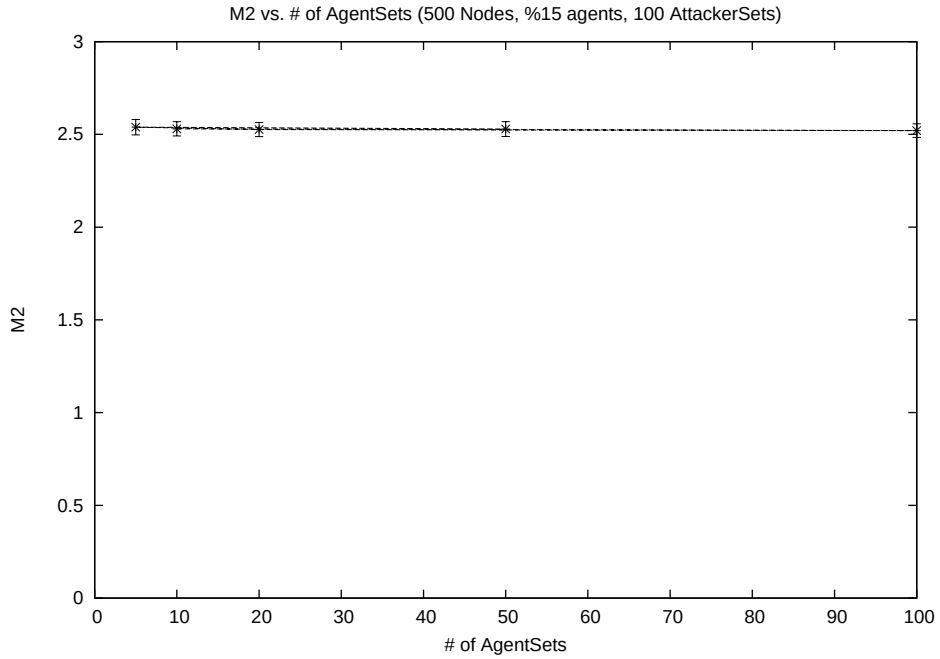


Figure 6.1.9: M2 versus number of agent sets

6.1.4 Experiment sensitivity to the number of attacker sets

Purpose. The purpose of this experiment is to investigate how the M1 and M3 performance measures are influenced by the number of attacker sets used for the experiment. Recall that we used 100 different attacker sets for the experiments 6.1.1 and 6.1.2. Upon completion of this experiment, we want to find out if the results of experiments 6.1.1 and 6.1.2 are valid in general or not. In another words, to what extent would the results be different if we had used a greater or smaller number of different attacker sets?

Experiment setup. The experiment operates as follows:

- Create a Waxman network $G = (V, E)$ of size 500 and let R be the routing table on G derived by Bellman-Ford's algorithm.
- Set the agent density to ϵ=15%.

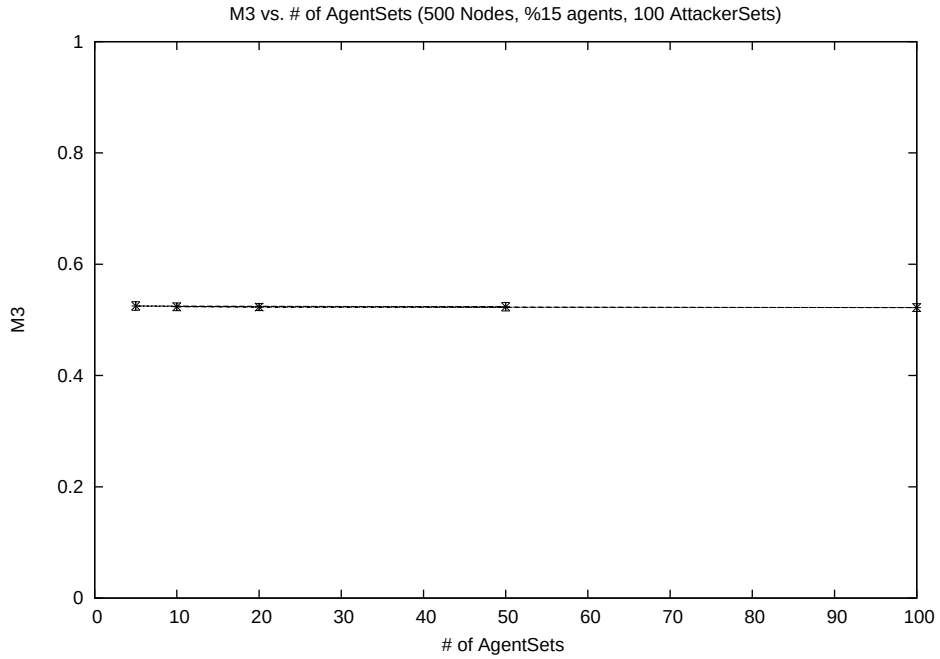


Figure 6.1.10: M3 versus number of agent sets

- Set the number of attackers to $|D| = 10$.
- For each of (100) agent sets of size ϵm, which are uniformly randomly selected.
- For each of ($\mathbf{Y}$) attacker sets of size δn, which are uniformly randomly selected.
- Consider each node v in V, in turn, as the victim.
- Create an instance FRP problem (G, R, A, D, v).
- Obtain a maximal valid solution and compute M1, M2, and M3 for the solution.

Z. Repeat the experiment for the set of $\mathbf{Y}$ values $(5, 10, 20, 50, 100, 500, 1000)$ while keeping the values of the rest of the variables the same.

Results. Figure 6.1.11 presents the dependence of M1 value on the number of attacker sets. The curve shows the mean value of M1 for different experiments with different number of attacker sets. The M1 curve shows a flat behavior as the number of agent sets increases.

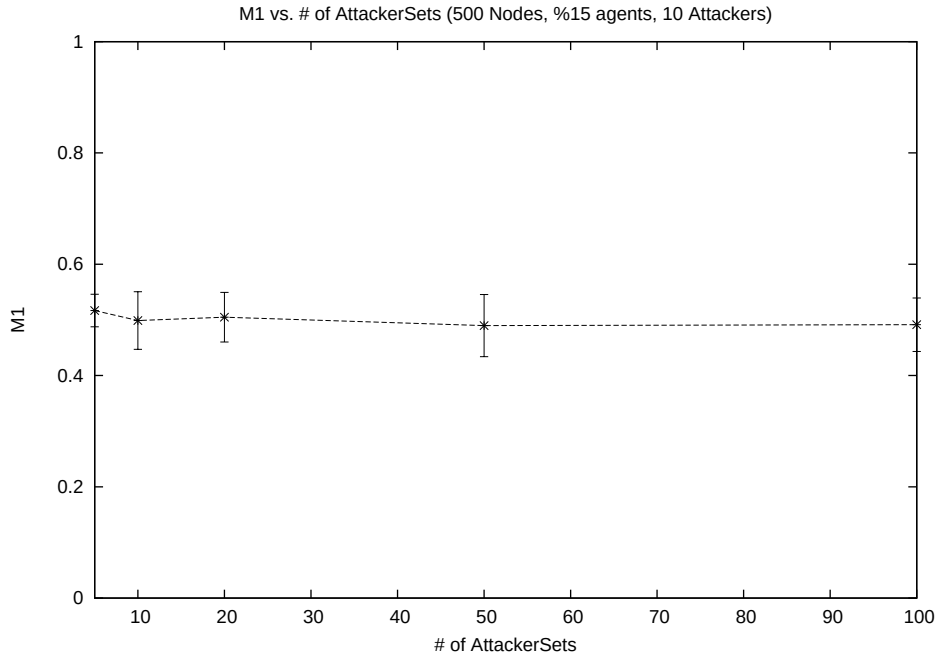


Figure 6.1.11: M1 versus number of attacker sets

The error bars shows the variances of the M1 values for each experiment. The average size of error bars is 0.02. The variance between the size of error bars are indistinguishable. Consequently, we say that the number of attacker sets does not influence the value of M1 significantly.

Figure 6.1.12 presents the dependence of M2 value on the number of attacker sets. The curve shows the mean value of M2 for different experiments with different number of attacker sets. The M2 curve shows a flat behavior as the number of agent sets increases. Similar to the case for M1, we say that the number of attacker sets does not influence the value of M2 significantly.

Figure 6.1.13 presents the dependence of M3 value on the number of attacker sets. The curve shows the mean value of M3 for different experiments with different number of attacker sets. The M3 curve shows a flat behavior as the number of agent sets increases. Similar

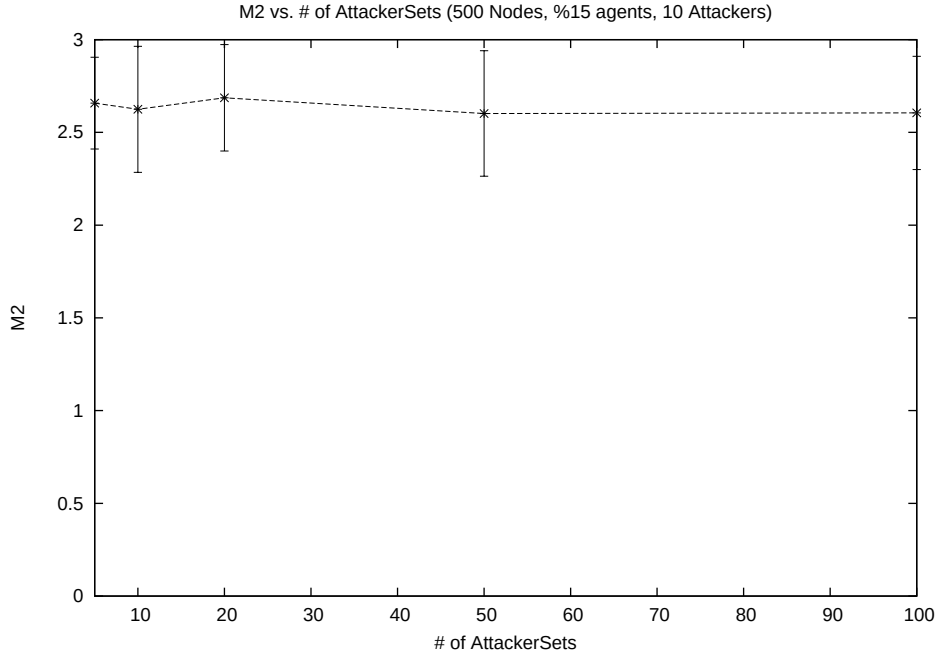


Figure 6.1.12: M2 versus number of attacker sets

to the case for M1 and M2, we say that the number of attacker sets does not influence the value of M3 significantly.

Figures 6.1.11, 6.1.12, and 6.1.13 clearly show that the performance values M1, M2, and M3 does not vary significantly in proportion to the number of attacker sets. Consequently, the results of experiments 6.1.1 and 6.1.2 are valid in general for different number of attacker sets.

6.1.5 Scalability for large networks

Purpose. The purpose of this experiment is to investigate how the performance measures scale with increasingly larger networks.

Experiment setup. For this experiment we did not run a new experiment instead we

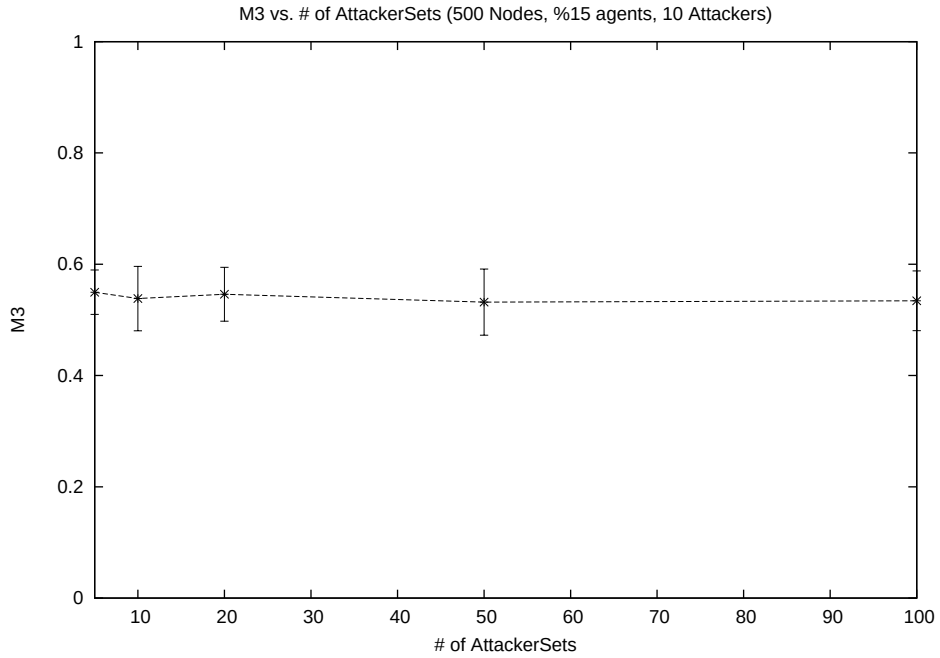


Figure 6.1.13: M3 versus number of attacker sets

used the data acquired from experiment 6.1.1.

Results. Figure 6.1.14 shows how M1 changes as the network size increases. Each curve considers a scenario in which agents are deployed with different densities (from a low of 1%, rising to 8%, 15%, 22% and finally 28%). The graph shows that we get better detection rate as the agent density increases and as the network size increases. However, the curve flatten implying that system performance metric M1 becomes almost independent of the network size, once the networks under consideration become sufficiently large (e.g. have more than 1000 nodes). In addition, the graph indicates that the M1 performance metric may be adequately estimated by running simulations of 1000 nodes; time-consuming simulations of larger networks are unlikely to exhibit significant changes in their M1 measure once agent density has been decided.

Figure 6.1.15 shows the dependence of M2 on network size. Once again, each curve in the

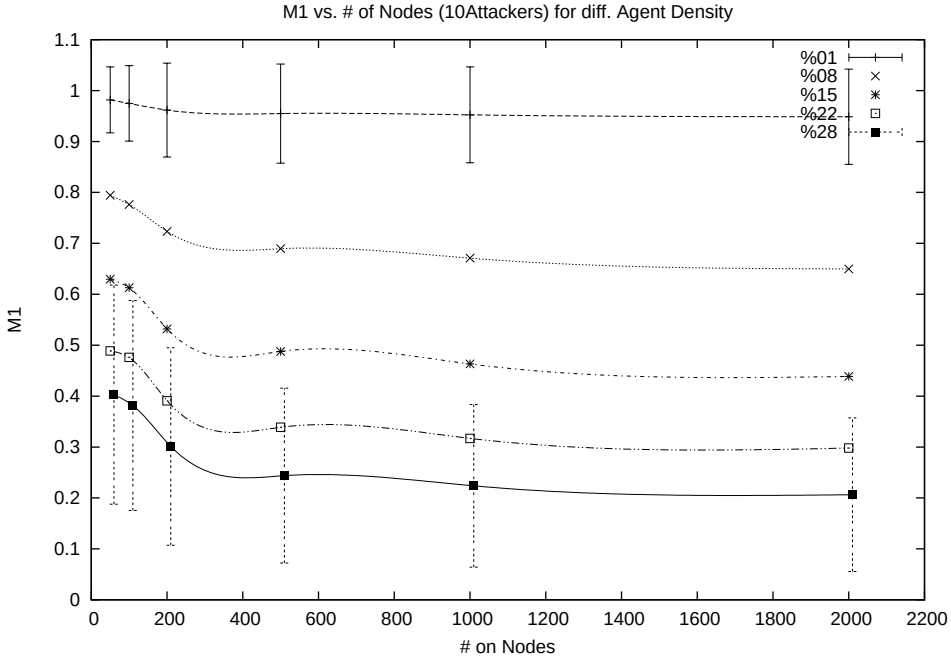


Figure 6.1.14: M1 versus number of nodes

graph considers a scenario in which agents have been deployed with different density. We see that that the average distance to the attacker increases as the network size increases, but that the increase in M2 is insignificant when compared to increases in the size of the network itself. M2 is also seen to decrease as agent density increases, but (observe that the curves are eventually parallel) the extent of this effect becomes largely independent of the network size. It is readily apparent that regardless of agent density, the curves plateau once the networks under consideration are sufficiently large (e.g. have more than 1000 nodes). Considering the current scale of the Internet and its 10,000+ ASes, the fact that we may be able to trace back DDoS attacks to within 2.5 hops of discovered attackers seems quite promising.

Figure 6.1.16 shows the dependence of M3 on network size. Once again, each curve in the graph considers a scenario in which agents have been deployed with different density. We see that that the normalized average distance to the attacker decreases as the network

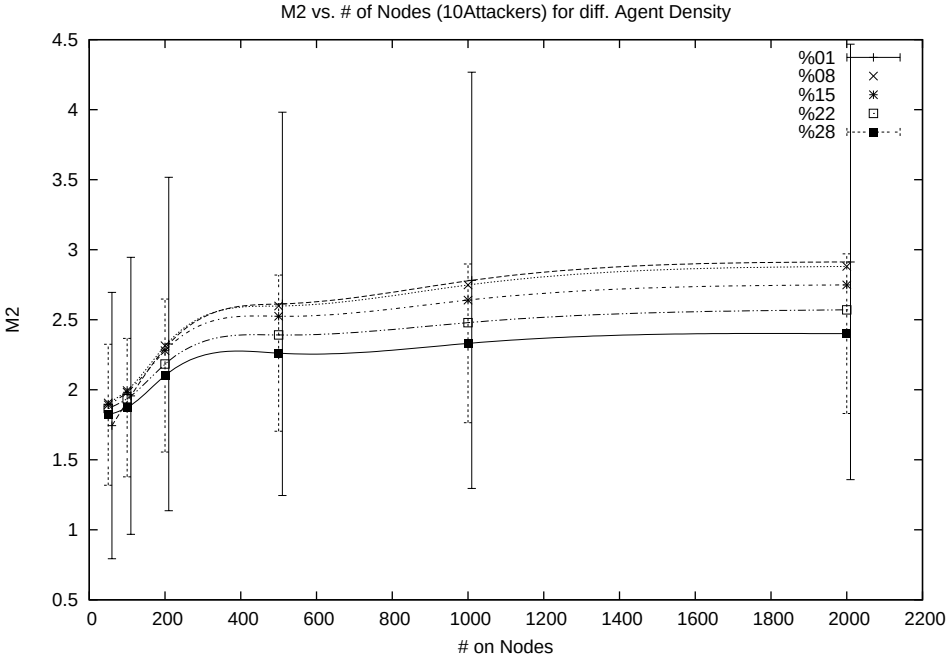


Figure 6.1.15: M2 versus number of nodes

size increases, but that the decrease in M3 is insignificant when compared to increases in the size of the network itself. M3 is also seen to decrease as agent density increases, but (observe that the curves are eventually parallel) the extent of this effect becomes largely independent of the network size. It is again clearly seen that regardless of agent density, the curves flatten once the networks under consideration are sufficiently large (e.g. have more than 1000 nodes). Again for large scale networks we may be able to trace back DDoS attacks approximately to the halfway towards the discovered attackers.

6.1.6 Experiments on the Internet topology

Purpose. The purpose of this set of experiments is to see how our agent based system performs on the real Internet topology. For this purpose we get AS information from CAIDA [68] project. We imported the CAIDA network in to our simulation, by creating

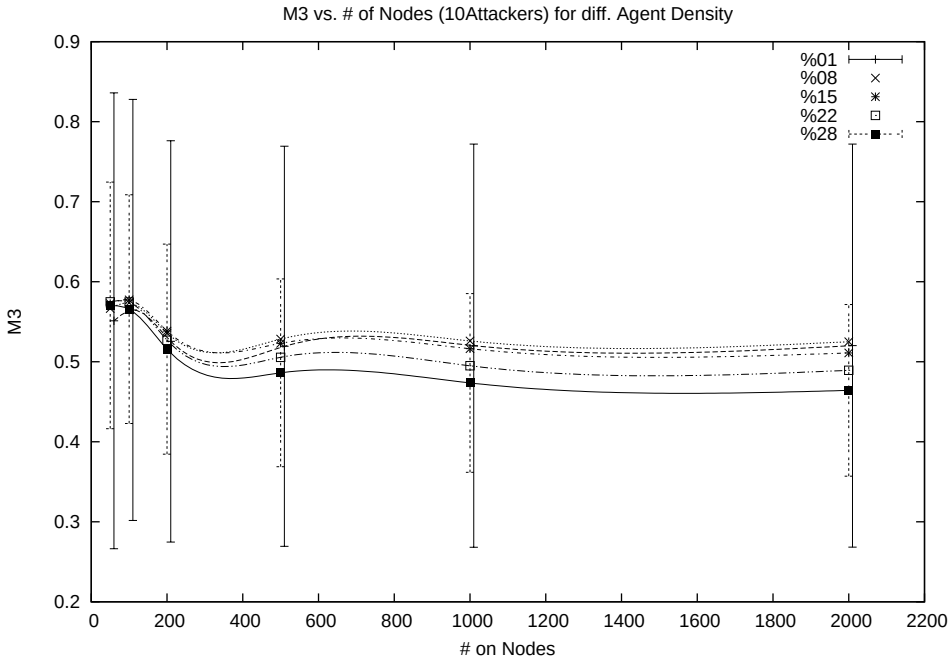


Figure 6.1.16: M3 versus number of nodes

the nodes and links between the nodes according to CAIDA data.

Experiment setup. For this part of the research we carried out 2 experiments. All the experiments are similar except network creating factories. We used our Waxman network factory to create network of size 200 nodes and CAIDA [68] network factory to import the CAIDA network and run experiment on it. CAIDA network factory gets a CAIDA AS adjacency data file and creates the corresponding nodes and links.

The first experiment is done for Waxman network with random agent placement. For the Waxman network experiment:

- For each agent density.
- Create a random agent set and repeat the following steps for ten times.
- For all possible attacker-victim pairs repeat the following.

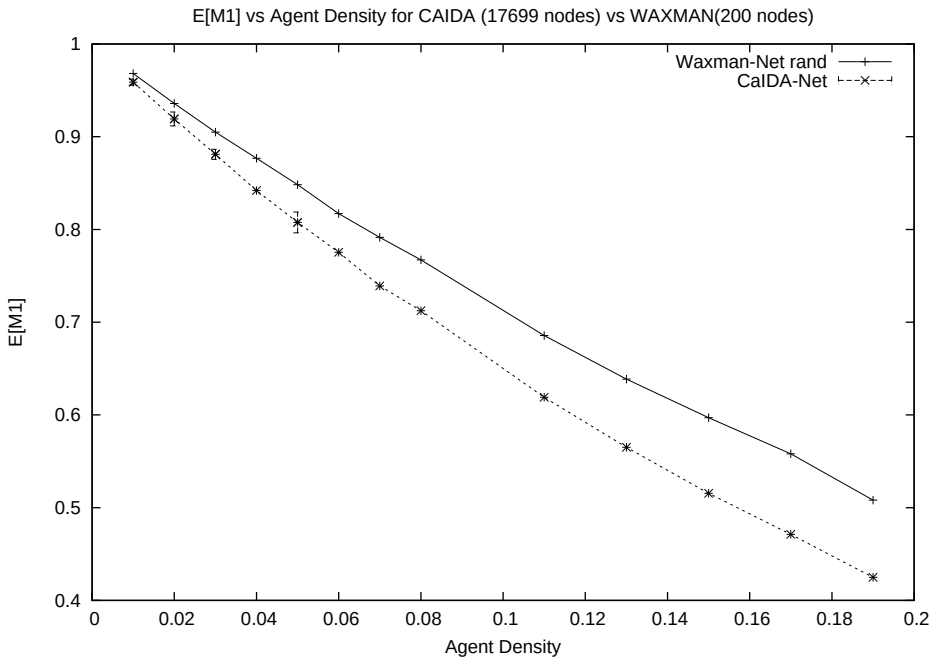
- Solve the problem and create $M1$ and $M3$ values.
- Calculate mean and standard deviation values after all possible attacker-victim pairs are completed.
- After repeat 10 times calculate the final mean and standard deviation for current agent density.

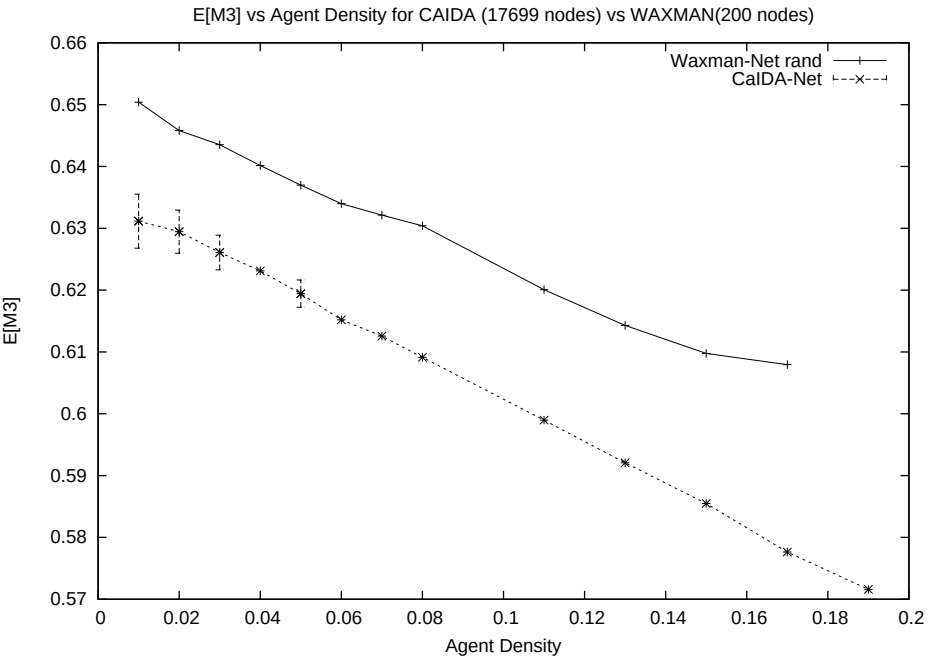
The second experiment is done for the CAIDA network. The experimental set up is exactly the same as the previous one except the CAIDA network is used.

Results.

Figure 6.1.6 shows M1 curve for random agent placement on two different networks. The first network is a Waxman network of size 200; the second one is the CAIDA network consisting of 17699 nodes. The graphs show us that the curves behaves similar as the agent density increases. Moreover, the curve for CAIDA network decreases faster then the random network.

The Figure 6.1.6 show M3 curve for random agent placement on two different networks. First network is a Waxman network of size 200 and the other one is the CAIDA network consisting of 17699 nodes. Graph show us that the curves behaves similar as the agent density increases. Both curves decreases parallel to each other. These results show that the performance of our system is similar in both randomly generated Waxman networks and on the real Internet topology.


E[M1] vs Agent Density for CAIDA (17699 nodes) vs WAXMAN(200 nodes)
Waxman-Net rand
CaIDA-Net
E[M1]
Agent Density
1
0.9
0.8
0.7
0.6
0.5
0.4
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
0.16
0.18
0.2


E[M3] vs Agent Density for CAIDA (17699 nodes) vs WAXMAN(200 nodes)
Waxman-Net rand
CaIDA-Net
E[M3]
Agent Density
0.66
0.65
0.64
0.63
0.62
0.61
0.6
0.59
0.58
0.57
0
0.02
0.04
0.06
0.08
0.1
0.12
0.14
0.16
0.18
0.2

6.2 Transient state measures

Recall that transient measures of our system includes traffic convergence time $t_c(v)$ and state convergence time $t_\sigma(v)$. We assume $t_c(v)$ converges to a value either greater or lower than the threshold T. The traffic convergence time can be controlled by the attackers. Consequently, for the rest of this section we will be interested in state convergence time CT that refers to $t_\sigma(v)$, which means the time it takes for our system to get stabilized.

We decided to use HTTP traffic pattern as the attack traffic pattern in our experiment, because attackers need to hide their attack traffic in order to be successful and we thought a mimicry attack (a type of attack which may allow attackers to fool the intrusion detection system by camouflaging attack so that it behaves much like a normal application would [62]) simulating a web traffic pattern would be a good way of hiding. We used constant size attack packets with inter arrival times chosen according to inverse Gaussian distribution (μ, λ) where μ represents the mean frequency of sending attack packets and λ represents the shape parameter. Because, according to Sun et.al [60] the inverse Gaussian distribution is the best fit for modeling inter arrival time of HTTP packets.

For each experiment, we created a connected network by randomly placing nodes in a 2D plane and randomly selecting links between nodes with respect their Waxman probabilities and built the routing table for the network. Afterwards, we create a problem to solve by randomly selecting attackers, agents, and a victim, where victim and attackers are members of routers and agents are members of physical links. After creating the FPR we imported it into our distributed event simulator. We implemented routers and agents as SimEnts and network packets as Events. We also created a report entity which serves as information collector.

Once simulation starts, agents start reporting their state changes to the experiment report entity. During the experiment, this report entity is checked regularly for changes. If there has been no change for significant amount of time, we decide that the agent system became stable and we than use this to calculate the CTs.

6.2.1 Experimental setup

The following pseudocode shows the overall execution of each simulation experiment:

- Set random number generator seed.
- Create an instance FRP problem (G, R, A, D, v) using centralized simulation code by.
  - Create a Waxman network, $G = (V, E)$, $|V| = n, |E| = m$ and let R be the shortest path routing table on G.
  - Set the agent density ϵ and attacker density δ.
  - Create an agent-set of size ϵm, which is uniformly randomly selected.
  - Create an attacker-set of size δn, which is uniformly randomly selected.
  - Select a node v in V, randomly, as the victim.
  - Create an instance FRP problem (G, R, A, D, v).
- Create the discrete version of given FRP by:
  - Creating router as a SimEnt for each $V \in G$.
  - Creating a physical link for each link $E \in G$.
  - Creating the agent set corresponding to $A \in FRP$.
  - Creating the attackers set corresponding to $D \in FRP$.

  - Creating the victim corresponding to $v \in FRP$.

- Start simulation by starting the Scheduler
- Start an attack on victim by sending fixed sized packets. The distribution of inter-arrival times of attack packets can be uniform or can follow inverse Gaussian distribution depending on the experiment.
- Report attack start time to the report entity.
- Check report entity regularly until there is no further state change in the report.
- Record convergence time during attack as the time of the last event added to the report minus attack start time.
- Stop attack.
- Report attack stop time to the report entity.
- Check report entity regularly until there is no further change in the report.
- Record convergence time after the attack as the time of the last event added to the report minus attack stop time.
- Calculate $M1$ and $M3$ for the distributed solution implemented by the logical connections between agents.
- Exit simulation

6.2.2 Experimental parameters

There are several variables used for the discrete simulation experiments. The generation of traffic is governed by the following parameters:

- μ is the **mean packet inter-arrival time**. On the average, every μ seconds, a new packet is send to the victim. In the CBR case, this is a deterministic periodic process. In the inverse Gaussian case, μ is used as the mean frequency, and λ is the shape coefficient.

- **Link-Delay** specifies the time it takes to deliver a packet from one node to another; it includes queuing and node processing times.

Agent measurement of traffic flows is governed by the following parameters:

- SCI is the **statistics collection interval**. Every SCI seconds, traffic statistics (per destination IP) are calculated by each of the agents[4].

- r is statistics history coefficient. It is the hysteresis coefficient governing the windowed average of traffic flowing towards a destination.

Agent reactions to traffic flows is governed by the following parameters:

- ATC is the **attack threshold coefficient**, and is typically a number $\geqslant 1$. Based on this, we define the attack threshold T to be $ATC * SCI/\mu$. An AH event is generated when traffic to the victim exceeds T. A BH event is generated when traffic to the victim falls below T.

Agent protocol parameters include:

- **Max-TTL** is the variable that shows the maximum value of TTL value that can be used for AD messages.

4. Note that in an attack on a node v, an agent implicitly expects to see SCI/μ packets per second.

- **ADA-COEFF** is used to calculate the agent's wait time (t_w) for an ADA message after it sends an AD message. It is defined to be

$$t_w = 2 * ADA - COEFF * LINK - DELAY * Last - TTL$$

  where last-TTL is the TTL value of the last AD message.

Randomization parameters include:

- **Random-Seed** is a long integer value used as a seed to the random number generator. Two different series of random numbers which are produced with the same random-seed follows exactly the same pattern.

There are several other parameters which we did not list them above because they were kept fixed for all the experiments. The list below provides their values:

- Number of nodes = 200
- Number of attacker = 10
- Number of different attacker sets = 1
- Agent density = 25%
- Number of different agent sets = 1
- Number of victims = 1

Through our experiments we seek the relationship between

- $M1$ and ATC,
- $M3$ and ATC,
- CTs and ATC,
- CT and $Link - Delay$, and
- CTs and r.

In order to do the experiment for CT versus r, we run the experiment by keeping everything fixed except the value of r. In all other experiments every parameter except the ATC values were kept fixed. We repeated the experiments for different Random-Seed values, to assess robustness of conclusions to different random of victims, attackers and agents.

6.2.3 Influence of ATC on $M1$ and $M3$

Purpose. The purpose of this experiment is to understand the influence of the attack threshold coefficient ATC on performance measures M1 and M3.

Experiment setup. We used the experimental setup defined in the beginning of this section. The parameter settings for this experiment were: $SCI = 1$, Max-TTL=8, Link-Delay=1, ADA-COEFF= 1.1, Attack packets' inter arrival time distribution = uniform, $\mu = 0.1$, $\lambda = NA$, $r = 0$. After setting these variables, we ran the same experiment for the following values of $ATC = [0 : 8]$ and $\epsilon = 0, 0.04, 0.08$.

Results. Figure 6.2.1 shows the relationship between $M1$ and ATC. As the value of ATC increases curves show stepping increase showing different patterns. An increase in the value of $M1$ means the number of undiscovered attackers has increased. This is a

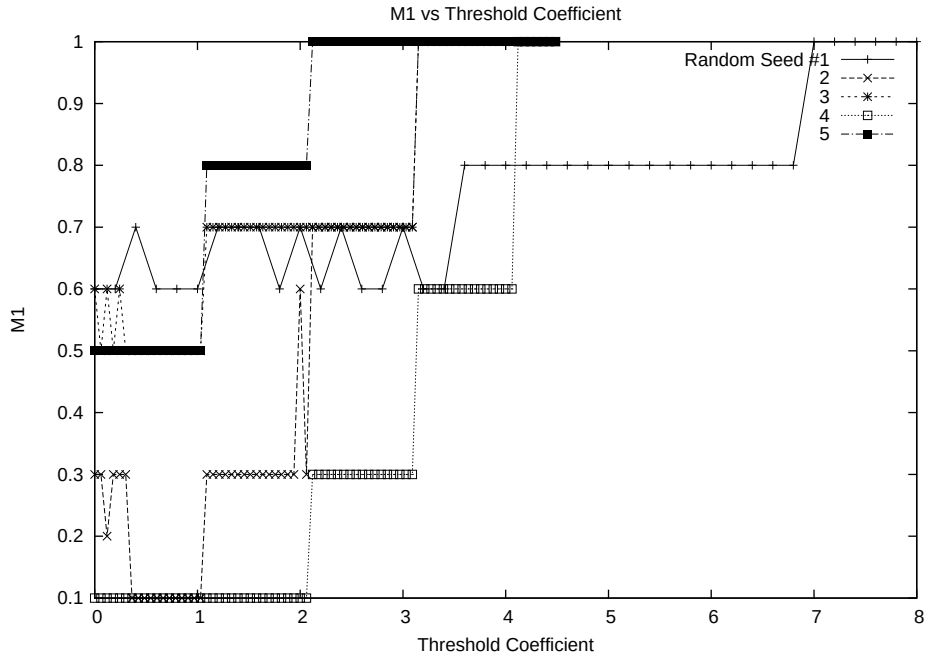


Figure 6.2.1: M1 as traffic exceeds thresholds

normal behavior because depending on their locations, the agents may experience traffic from different number of attackers and this may result in seeing different amount of attack traffic. Consequently, T will be greater than the attack traffic sent from single attacker and any agent seeing attack traffic from single attacker will not detect the attack anymore because the attack traffic it sees will be less than T. If there is no more agent down on the path of the attack, the corresponding attacker will not be discovered, so $M1$ will increase. Similar behavior is repeated as T passes the multiples of estimated attack traffic value. We also see that there are four interesting points (approximately at 3.5, 7, 10.5 and 14) where the curves experience a stepping increase. At these points, the value of T is approximately equal to the multiples of estimated attack traffic to v. Initial results show that threshold selection has a direct effect on $M1$.

Figure 6.2.2 shows the relationship between $M3$ and ATC. As the value of ATC increases curves show stepping changes up and down with different patterns. An increase in the

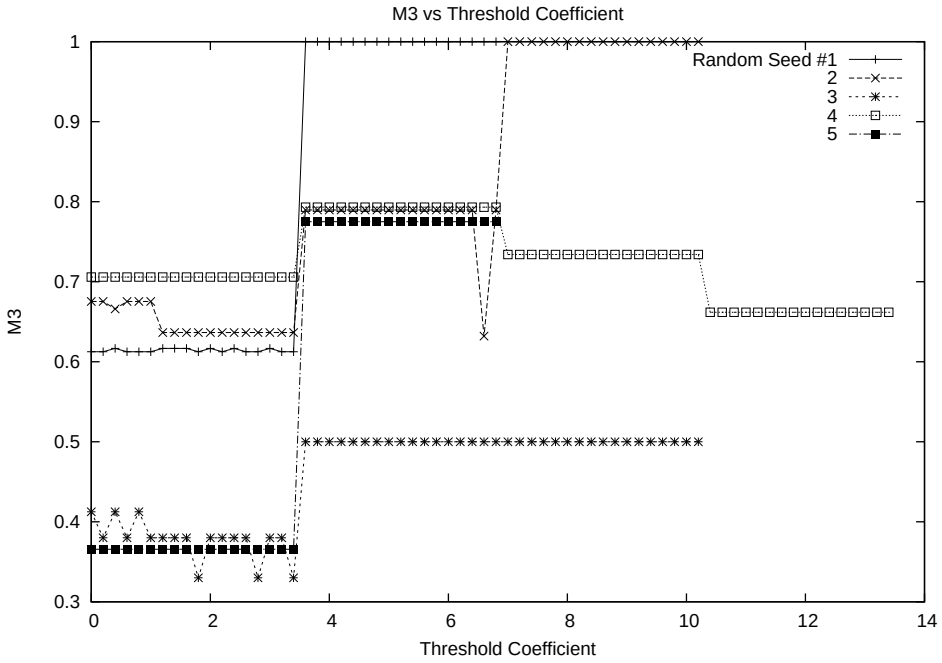


Figure 6.2.2: M3 versus threshold

value of $M3$ means the normalized distance to a discovered attacker has increased. This is a normal behavior because depending on their locations, the agents may experience traffic from different number of attackers and this may result in seeing different amount of attack traffic. Consequently, T will be greater than the attack traffic sent from single attacker and any agent seeing attack traffic from single attacker will not detect the attack anymore because the attack traffic it sees will be less than T. If there is no more agent down on the path of the attack, the corresponding attacker will not be discovered, so $M3$ will increase. Similar behavior is repeated as T passes the multiples of estimated attack traffic. Unlike $M1$ curve, one of the $M3$ curves decreases. This behavior is also normal, because recall that $M3$ just considers the detected attackers. Consequently, in cases when threshold increase causes disappearance of attackers which were contributing badly to the $M3$ value, the resulting $M3$ value will decrease. We also see that there are four interesting points (approximately at 3.5, 7, 10.5 and 14) where the curves experience a stepping increase. At these points, the value of T is approximately equal to the multiples of estimated attack

traffic to v. Initial results show that threshold selection has a direct effect on $M3$.

6.2.4 Influence of ATC on state convergence time

Purpose. The purpose of this experiment is to understand how the attack threshold coefficient ATC influences the state convergence time of our agent based system, immediately after an attack starts or after an attack stops.

Experiment setup. We used the experimental setup defined in the beginning of the section. Following are the values of variables that are used for this experiment: $SCI = 1$, $\epsilon = 0$, Max-TTL=8, Link-Delay=1, ADA-COEFF= 1.1, Attack packet inter arrival time distribution = uniform, $\mu = 0.1$, $\lambda = NA$, $r = 0$.

After setting these variables, we run the same experiment the following values of $ATC = [0 : 4.5]$ and random seeds from $\{1, 2, 3, 4, 5\}$.

Results. Figure 6.2.3 shows the relationship between CT and ATC. There are these curves with different epsilon values. We will start with the curves with epsilon value of zero. Results show that the agent system gets stabilized for certain values of ATC and never gets stabilized for others. Curves show that the CT are approximately same for the values of ATC less than 3. After this point CT go to zero which means no attacker is detected because the value of T was too high.

Although it is expected that there is a decrease as the value of T increases, the system structure of agents forces them to wait for certain amount of time to decide weather they have a child or not and than change their state to their final stable state. This results in not seeing remarkable decreases in the value of CT during the attack.

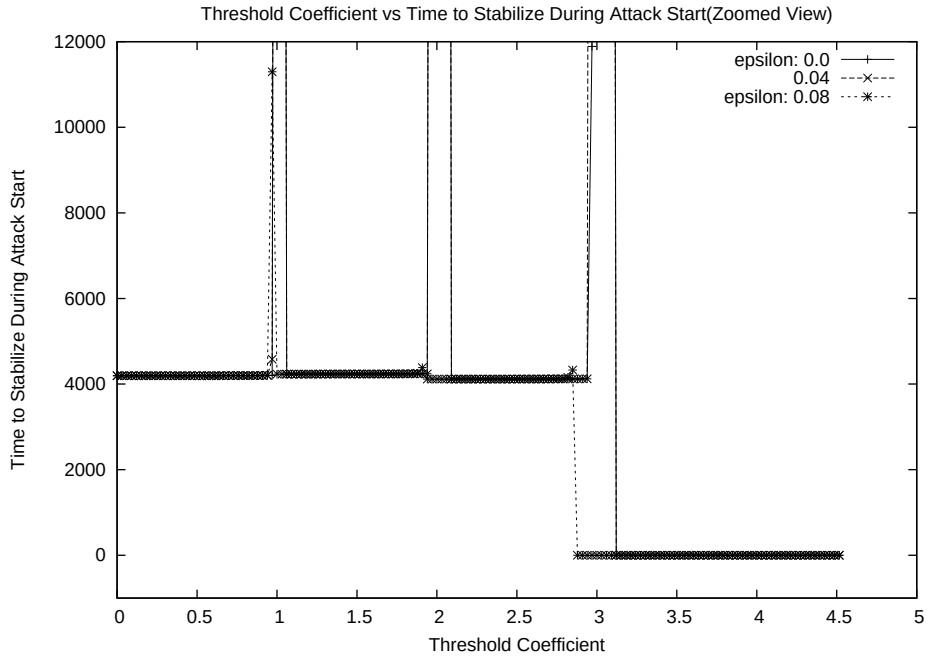


Figure 6.2.3: System convergence time at attack start

Similarly, in figure 6.2.4 we see the relation ship between the CT and ATC after attack has stopped. Curves show a stepping decrease as the value of T increases. This is because the number of agents in the solution set decreases as the value of T increases. In attack stop case all agents quit from the solution set when they think they should. The results are different than the previous results because when an agent decides to quit from the solution set it does it right away without waiting after it informs its peers. In the other case agents have to wait for certain amount of time for a response from possible pairs. Consequently, the system gets stable faster in this case since agents change their states as soon as the traffic goes below T and there is no wait time.

In figure 6.2.3 and figure 6.2.4 the curves show peaks at certain points. The peaks goes to infinity for the first case and a remarkably higher value than their previous values in the latter case. These peaks occur because the estimated traffic value and the value of T are very close to each other and since we use variable inter packet difference the estimated

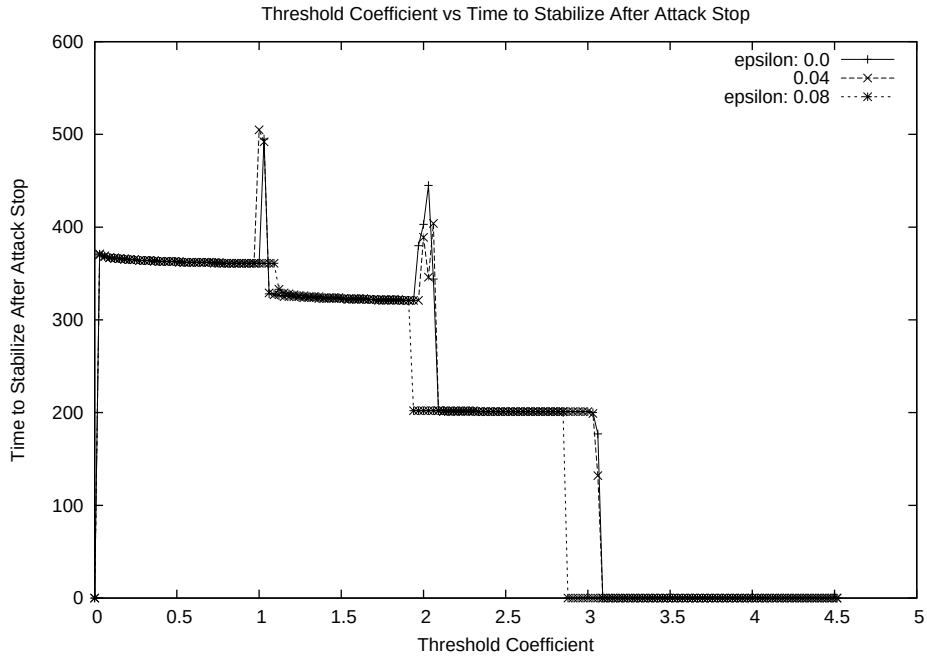


Figure 6.2.4: System convergence time at attack stop

traffic values fluctuates over T which results in not getting stable traffic and as a result no stable state. This happens since we have a fixed T.

Since this kind of behavior is undesired, we modified our system introducing a new parameter to govern agent reactions to traffic flows:

- ϵ is the **margin** which governs the generation of events, $\epsilon > 0$. Now, an AH event is generated when traffic to the victim exceeds $(1 + \epsilon)T$. A BH event is generated when traffic to the victim falls below $(1 - \epsilon)T$.

After modification of the system we repeated the experiments for epsilon values of 0.04 and 0.08. In figure 6.2.3 and figure 6.2.4 we see that the spikes decreased as we used greater epsilon. Both figures show that when we increase ϵ from 0 to 0.08 the spikes disappear. This shows that the system can made to respond in a stable manner at all attack levels.

6.2.5 Influence of traffic history coefficient on state convergence time

Purpose. The purpose of this experiment is to explore the relationship between state convergence time CT and traffic history coefficient r.

Experiment setup. We used the experimental setup defined in the beginning of the section. The following parameter values were used for this experiment: $SCI = 1$, $\epsilon = 0$, $Max - TTL = 8$, $Link - Delay = 1$, $ADA - COEFF = 1.1$, Attack packet inter arrival time distribution = uniform, $\mu = 0.1$, $\lambda = NA$,$ATC = 1$. After setting these variables, we run the same experiment the following values of $r = [0 : 1)$ (incremented in steps of 0.1).

Results. Figure 6.2.5 shows the relation between CT and r during the attack start. There are 5 different curves shown in the graph. Each figure corresponds to a different FRP

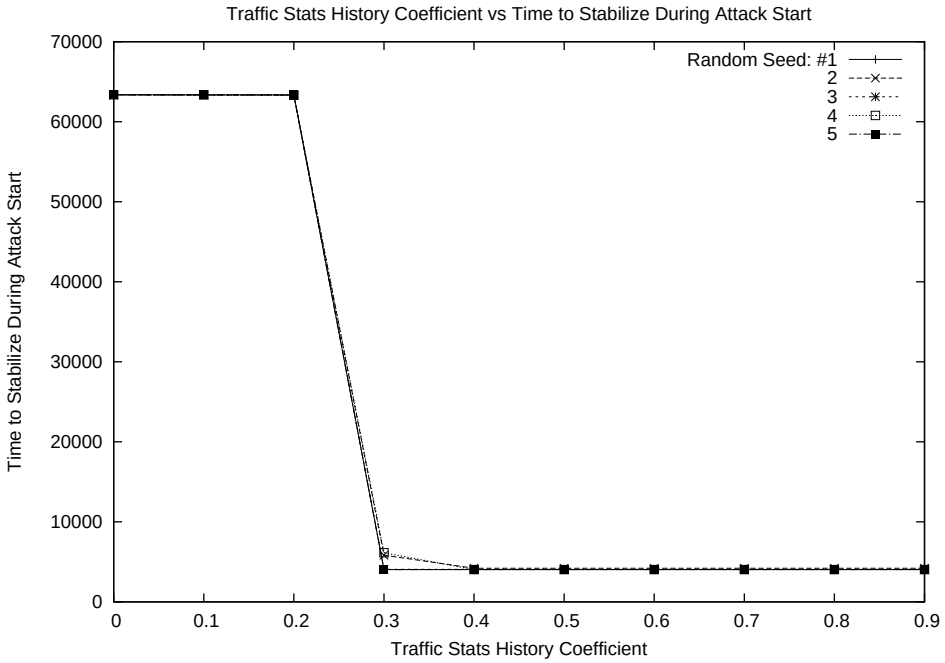


Figure 6.2.5: THC versus StStart

problem. All the curves except one show exactly same pattern. The last curve is almost the same too. This behavior shows that our system has a fixed CT for different FRPs.

Curves decline for all the curves, the CT value is high when the r is between 0 and 0.1. At 0.1 the curve for random seed "2" shows a 30% decrease and then stays flat. Rest of the curves declines almost 90% after r becomes "0.3" and then stays stable. Figure 6.2.5 shows that the smaller the value of r the more we are effected by the by the traffic fluctuations. It also shows that the selection of r does have a huge effect on CT. We can also concludethat $r = 0.3$ would be the good value for the rest of the experiments.

Figure 6.2.6 shows the relation between CT and r after attack stopped. There are 5 different curves shown in the graph. Each figure corresponds to a different FRP problem.

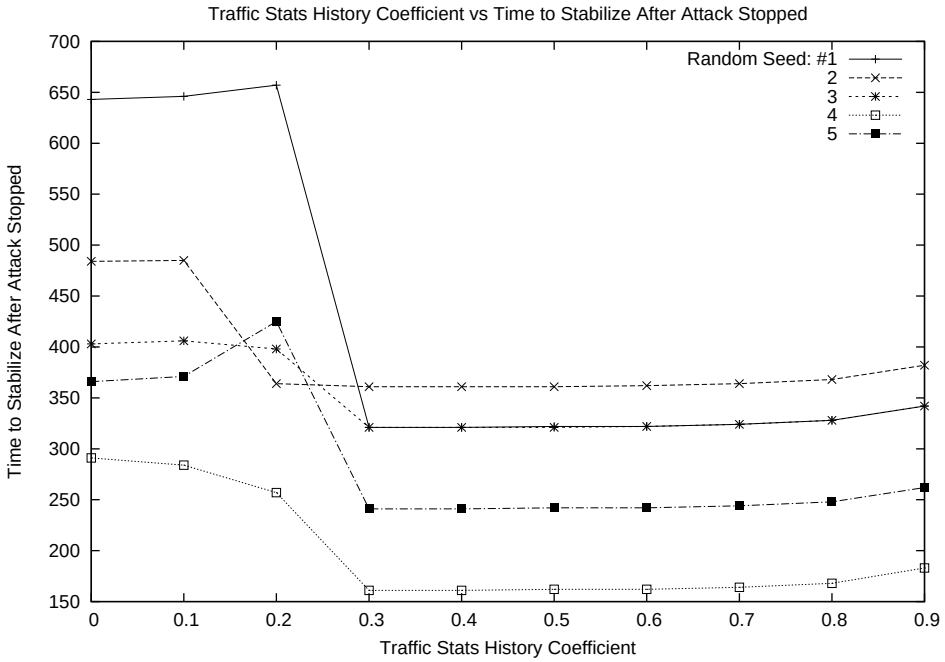


Figure 6.2.6: Hysteresis in detecting attack stop

For all the curves, the CT value is high when the r is between 0 and 0.1. At 0.1 the curve for random seed "2" shows a 30% decrease and then stays flat. Rest of the curves declines after r becomes "0.2" and then stays stable. Figure 6.2.6 shows that the smaller the value of r the more we are effected by the by the traffic fluctuations. It also shows that $r = 0.3$ would be the good value for the rest of the experiments.

6.2.6 Influence of link-delay on state convergence time

Purpose. The purpose of this experiment is to explore the relationship between state convergence time CT and link delay.

Experiment setup. We used the experimental setup defined in the beginning of the section. The following values were used for the parameters: $SCI = 1$, $\epsilon = 0$, $Max-TTL = 8$, $ADA-COEFF = 1.1$, Attack packet inter arrival time distribution = uniform, $\mu = 0.1$, $\lambda = NA$,$ATC = 1$, $r = 0.3$. After setting these variables, we run the same experiment the following values of , $Link - Delay = [0 : 100]$ (in increments of 1).

Results. Figure 6.2.7 shows the relation between $Link - Delay$ and CT during attack start. There are seven curves each of which represents a different FRP. All seven curves show the same type of behavior. The CT curve increases linearly Link-Delay increases.

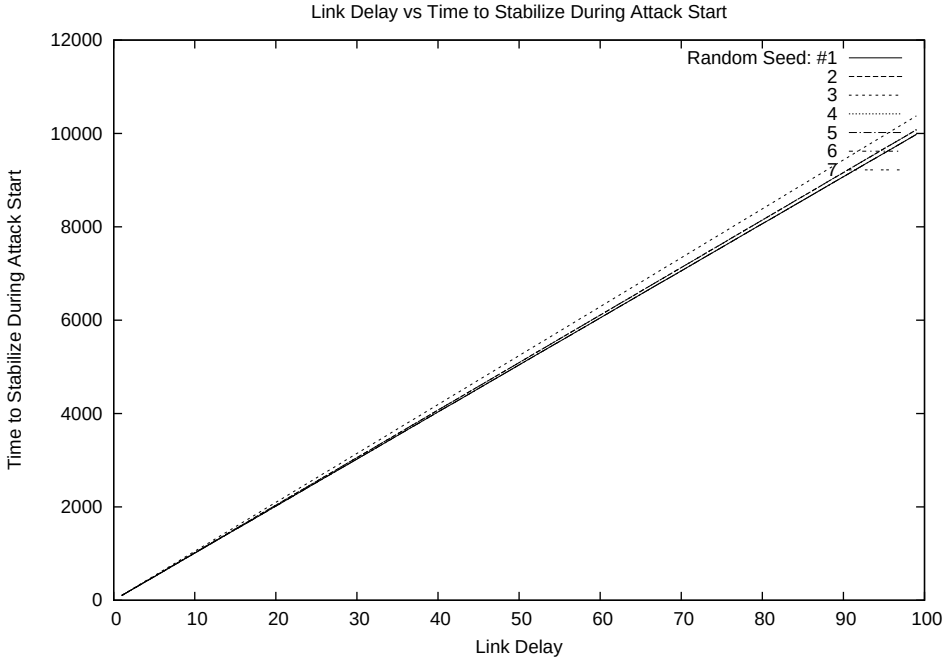


Figure 6.2.7: The effects of Link-Delay during attack start

This shows that the Link-Delay has direct effect on CT. This is an expected situation

since the agent communication directly depends on the network communication. However, all the curves' being almost the same is not an expected situation. This shows that the CT of our agent system does behave the same for different situation. However, we cannot say that it is completely independent of the FRP since one of the lines emerges from the others as we move further right on x-axis. We can also conclude that the CT during attack start dominated by the ADA wait time. Consequently, we can say that the CT during attack start depends linearly on link delay, but negligibly low on the FRP.

Figure 6.2.8 shows the relation between Link–Delay and CT after attack stopped. There are seven curves each of which represents a different FRP. All seven curves show the same behavior. Unlike the case during attack start, the curves show five different paths. Lines

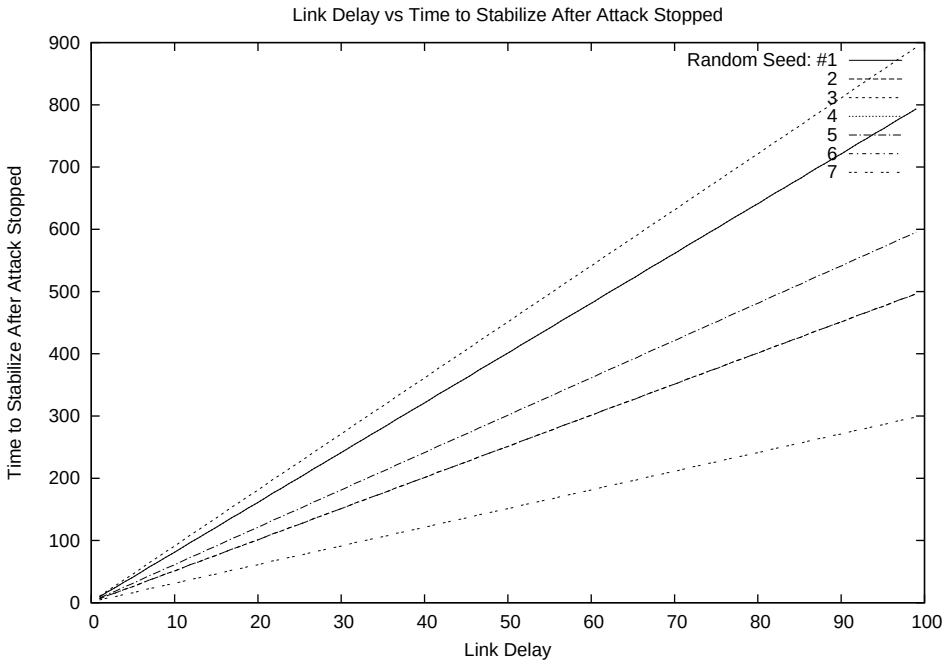


Figure 6.2.8: The effects of Link–Delay during attack stop

from top to bottom have slopes 9, 8, 6, 5, and 3 respectively. Recall that during attack stop, agents just leave the solution set to become stable. Before leaving the set they inform their peers. Informing a pair at one hop requires $1 \times Link - -Delay$ seconds. If there

are 3 agents constituting a path which are one hop apart from each other than it will take $3 \times Link - -Delay$ for the system to become stable. The value of the top most line at Link-Delay 100 is 900. This means it took $9 \times Link - -Delay$ for the system to become stable. From this we can assume that in the worst case there were two agents which are 9 hop apart from each other. Consequently, we can say that the CT during attack stop depends linearly on link delay however, its slope is determined by the agent solution set of the recently ended attack.

6.2.7 Summary

Through simulations, we showed that a significant fraction of attackers can be discovered even with very modest deployments of agents. We are able to get (on average) within 2-3 hops from the attackers. By providing such structural information, the proposed system can facilitate mitigation strategies close to the actual sources of attack traffic. In addition, we were able to quantify the transient characteristics of our novel distributed flow reconstruction system. Results of the simulations showed that there are certain scenarios in which the system protocol states do not converge. We were able to isolate the parameter ranges in which this phenomenon is manifested, with the help of simulations. By analyzing these problematic scenarios further, we were able to discover the responsible mechanisms, and modify our system to address these stability issues. Upon re-executing the experiments for the updated system, we verified that the state convergence is manifested over the entire range of parameter values.

CHAPTER 7

EXTENSIONS: AGENT PLACEMENT OPTIMIZATION

We saw in previous chapters that the proposed agent system is scalable, and capable of reconstructing traffic flow structure even under assumptions of modest agent deployment levels. In this Chapter, we extend the previously described system. The starting point of the extension considered is the large variances in the experiments involving many random agent placements (see Chapter 6). This phenomenon implies that the system's performance with respect to M1 and M3 measures depends greatly on the placement of agents–some random placements produced good performance, while others performed poorly. The question naturally arises: Is there any way in which the agents in our system can be *deliberately* placed so that our system performance is, on average, better?

What exactly is a good or bad placement? A badly placed agent, intuitively, would be one that has been "Wasted" by being placed on a link which is unlikely to witness attack flows. Conversely, a good placement of agents would be one in which each agent is expected with high probability to witness attack flows, presuming no prior knowledge of attacker and victim location. In what follows, we formalize this intuition and use it to devise algorithms for placing agents within a network. In this chapter we demonstrate the extent to which the proposed placement algorithms are capable of at improving the effectiveness of the agent-based flow reconstruction system.

7.1 Optimal placement

Before going further, we need to have a sense of how difficult it is to compute the optimal solution to the agent placement problem. As noted in Chapter 6, there are at least two reasonable ways to define this: The M1 metric and the M3 metric. Here we will consider optimality with respect to M1.

Let P a program that solves the flow reconstruction problem (G, R, A, D, v) and produces a maximal valid solution[1]. Based on this, we can define

Definition 7.1.1. *The **average percentage of discovered attacks**(U) for given $\delta > 0$ is*

$$U(P, G, R, \delta, A) = \frac{\sum_{v \in V} \sum_{(D \subseteq V) \& |D| = \delta|V|} M1(P(G, R, A, D, v))}{|V| * (|V| \in \delta|V|)}$$

Intuitively, $U(P, G, R, \delta, A)$ is the expected value of M1 achieved by P, for a fixed network G, routing table R and agent set A–over all attacker sets of size $\delta|V|$.

An upper bound on the complexity of computing U can be determined as follows: The inner most algorithm is P which will determine a solution to flow reconstruction problem. We can assume that computing M1 of a solution takes constant time. The complexity of the sum over the sets of cardinality $\delta|V|$ is, in general, as big as the cardinality of power set which is $O(2^{\delta|V|})$. Finally the complexity of the outer most function is $O(|V|)$. Consequently the overall complexity of U is going to be dominated by $O(2^{|\delta V|})$ as $|V|$ gets bigger. As a result the naive computation of U grows exponentially with network size.

We can use the expression defining U as a starting point for defining the best possible value

1. Note that there is unique maximal valid solution, and hence a (functionally) there is a unique program P.

of U attainable by program P, that is, optimal over all agent sets A of size $\epsilon|E|$. This is simply:

$$\overline{U}(P, G, R, \delta, \epsilon) = \min_{U}\{U(P, G, R, \delta, A) \mid (A \subseteq E) \wedge (|A| = |\epsilon|E||\}.$$

Now, given $\overline{U}$, let $A(P, G, R, \delta, \epsilon) \subseteq E$ be the set of agents for which the minimum was achieved

$$(P, G, R, \delta, A^*) = \overline{U}(P, G, R, \delta, \epsilon).$$

An upper bound for the complexity of computing $\overline{U}$ and $A(P, G, R, \delta, \epsilon)$ can be derived by computing U for all cardinality subsets of E, which may be $O(2^{|E|}$ in number.

It is clear from the above discussion that a naive brute force computation of $A(P, G, R, \delta, \epsilon)$ is not feasible. More precisely, it is nontrivial to find an algorithm which takes as input a network G, routing table R, and assumptions about attacker (resp. agent) density δ (resp. ϵ), and computes the placement of agents that minimizes the expected number of undiscovered attackers. Such an algorithm is likely to operate at a complexity that is exponential in $|V|$.

7.2 Evaluating an agent placement

We note that the two performance measures $E[M1]$ and $E[M3]$ defined in Section 4.4.2 were functions of only the network $G = (V, E)$, the routing table R, and the agent set $A \subset E$. Thus, for a fixed network and routing table, these two measures $E[M1]$ (see eqn. 4.1, page 93)) and $E[M3]$ (see eqn. 4.2, page 93) can serve to differentiate between different agent sets.

More concretely, given two equinumerous agent sets $A_1, A_2 \subset E$ for which $|A_1| = |A_2| = n$, an assertion like

$$E[M1](G, R, A_2) > E[M1](G, R, A_1)$$

can be interpreted as expressing the fact that placing n agents according to the specification A_1 yields a lower fraction of undetected attack flows than placing the agents according to specification A_2. The placement A_1 is thus better.

Now suppose we have a third agent placement A_3 of n agents (i.e. $|A_3| = n$) for which $E[M1](G, R, A_1) = E[M1](G, R, A_3)$ but

$$E[M3](G, R, A_1) > E[M3](G, R, A_3)$$

This would imply that the two agent placement schemes A_1 and A_3 discover the same fraction of attack flows, but in placement A_3 the normalized distance from intercepting agents to attack flow sources is smaller. Placement A_3 is better than placement A_1.

7.3 Greedy agent placement algorithms

We will describe two deterministic algorithms for placing agents in a network G (with routing table R). These algorithms are namely *M1-Greedy* and *M3-Greedy*. We will compare their performance relative to the expected performance of an adversary named *Random*, which places agents randomly on network edges. In the next sections we will describe each algorithm in detail.

Random. The Random agent placement algorithm serves as a baseline against which to compare our agent placement schemes. The Random placement algorithm takes as input the network $G = (V, E)$, the routing table R, and the number of agents n which are to be placed. It operates by randomly selecting an edge $e \in E$ which does not already have any agent on it, and places an agent on that link e. This is repeated until the desired number of agents have been placed in G.

M1-Greedy Algorithm. The M1-Greedy algorithm sequentially places the specified number of agents, one by one, in a manner that greedily minimizes $E[M1]$ at each step. Suppose agents $1, ..., i-1$ have been placed already. The algorithm places agent i as follows:

- Create a map from E to natural numbers; initialize all entries to 0.
- For all possible attacker-victim pairs in $V \times V$, do the following:
  - Start from attacker, proceed hop by hop according to R. At each hop, if there is no agent on the path increment the number associated with all the edges in the path by 1. If there is an agent on the edge, continue on to the next attacker-victim pair.
- Return the edge associated with the highest value as the placement for the next agent and add an agent on that link.

The Worst Case Complexity of M1-Greedy Algorithm: In order to place a single agent, current implementation of M1-Greedy algorithm does the following:

Assume $A(V) : V \times V \rightarrow \mathbb{N}$ represents the average number of hops between two nodes in $G(V, E)$.

- For all possible attackers (APA) ($|V|$),
- For all possible victims (APV) ($|V-1|$),
- For the average number of hops between any two nodes ($A(V)$),
- Get next hop from hash map. According to Java tutorial [41] the complexity of retrieving information from a hash map has constant-time performance.
- Check agent set, which is implemented as a hash set, if there is any agent in between current node and next-hop node. According to Java tutorial [42] the complexity of basic operations on a hash set has constant-time performance
- Get current value of the link from hash map. According to Java tutorial [41] the complexity of retrieving a value from a hash map has constant-time performance
- Put new value of the link to the hash map. According to Java tutorial [41] the complexity of adding a value to hash map has constant-time performance

Consequently, the complexity of M1-Greedy to place a single agent becomes

$$APA \times APV \times A(V) \times (get + check + get + put)$$

where number of all possible attackers is $APA = |V|$, the number of all possible victims is $APV = |V-1|$, the average number of hops between two nodes is $A(V) = \frac{\sqrt{|V|}}{3}$, and the rest of the operations has constant-time value. Consequently the complexity of M1-greedy to place a single agent becomes $|V| \times |V-1| \times \frac{\sqrt{|V|}}{3} = \mathcal{O}|V|^{2.5}$.

M3-Greedy Algorithm. The M3-Greedy algorithm sequentially places the specified number of agents, one by one, in a manner that greedily minimizes $E[M3]$ at each step. Suppose agents $1, ..., i-1$ have been placed already. The algorithm places agent i as follows:

- Create a map from E to real numbers; initialize all entries to 0.
- For all possible attacker-victim pairs in $V \times V$, do the following:
  - Start from the attacker, and proceed hop by hop according to R. Find the first agent f on the flow from the attacker towards the victim. On each edge e that lies on the flow from the attacker to f, compute the reduction in M3 that would be obtained by placing agent i on e. Increment the number associated with e by the magnitude of the reduction obtained.
- Return the edge associated with the highest value as the placement for the next agent and add an agent on that link.

The Worst Case Complexity of M3-Greedy Algorithm: In order to place a single agent, current implementation of M1-Greedy algorithm does the following:

- For all possible attackers (APA) ($|V|$),
- For all possible victims (APV) ($|V-1|$),
- For the average number of hops between any two nodes ($A(V)$),
- Get next hop from hash map.
- Check agent set if there is any agent in between current node and next-hop node.
- Get current value of the link from hash map.
- Put new value of the link to the hash map.
- For the average number of hops between any two nodes,
- Get next link's temporary benefit value from hash map.

- Compare temporary benefit value and current edge's benefit
- Put new benefit information in to the hash map.

Consequently, the complexity of M3-Greedy to place a single agent becomes

$$APA \times APV \times A(V) \times (get + check + get + put) \times A(V) \times (Get + compare + put)$$

where number of all possible attackers is $APA = |V|$, the number of all possible victims is $APV = |V - 1|$, the average number of hops between two nodes is $A(V) = \frac{\sqrt{|V|}}{3}$, and the rest of the operations has constant-time value. Consequently the complexity of M3-greedy to place a single agent becomes $|V| \times |V - 1| \times \frac{\sqrt{|V|}}{3} \times \frac{\sqrt{|V|}}{3} = \mathcal{O}|V|^3$.

7.4 Experiments

Each experiment proceeds as follows.

- Set the network size $|V|$ and create Waxman network G of this size. This is done by taking a geometric square of side $5\sqrt{|V|}$. Within this we place $|V|$ nodes uniformly at random. This establishes the vertex set V. Then we add random links between nodes, chosen with probability that is inversely proportional (exponentially) to the Euclidean distance of two nodes. This link addition process continues until the desired edge density is reached. Once the desired edge density is reached, we check if the network is connected or not. If it is we return the network. Otherwise, we continue to add edges until the graph becomes connected. In our experiments, we took the edge density to be slightly above two (2.1); this value was chosen because it is close to the value obtained through analysis of the present Internet topology [68]. This

establishes the edge set E. In this way, we have constructed a graph $G = (V, E)$ which has the form of a Waxman network whose edge density reflects present edge densities in the Internet.

- Instantiate a routing table R for this network by running Dijkstra's algorithm from all nodes.

- For each agent density ϵ:

  - Create the agent set A, where $|A| = \epsilon|E|$ using the M1-Greedy and M3-Greedy agent placement algorithms.

  - For all $(|V|^2)$ possible attacker-victim pairs (attacker = x, victim = v) repeat the following:

    * Solve the maximal valid solution to the FRP problem $(G, R, A, D = \{x\}, v)$ and compute the $M1$ and $M3$ values of the problem-solution pair.

  - Calculate mean and standard deviation over $(|V|^2)$ possible attacker-victim pairs, for the computed M1 and M3 values. We then plot E[M1] and E[M3] as a function of the agent density ϵ, and indicate the variance of M1 and M3 using error bars on the same curves.

7.4.1 Impact of agent density on performance

Purpose. The purpose of the first experiment is to quantify how the two agent placement algorithms perform at comparable deployment levels, both relative to each other, and relative to the expected performance of the random placement scheme.

Experiment setup. For this part of the research we carried out three experiments. Each

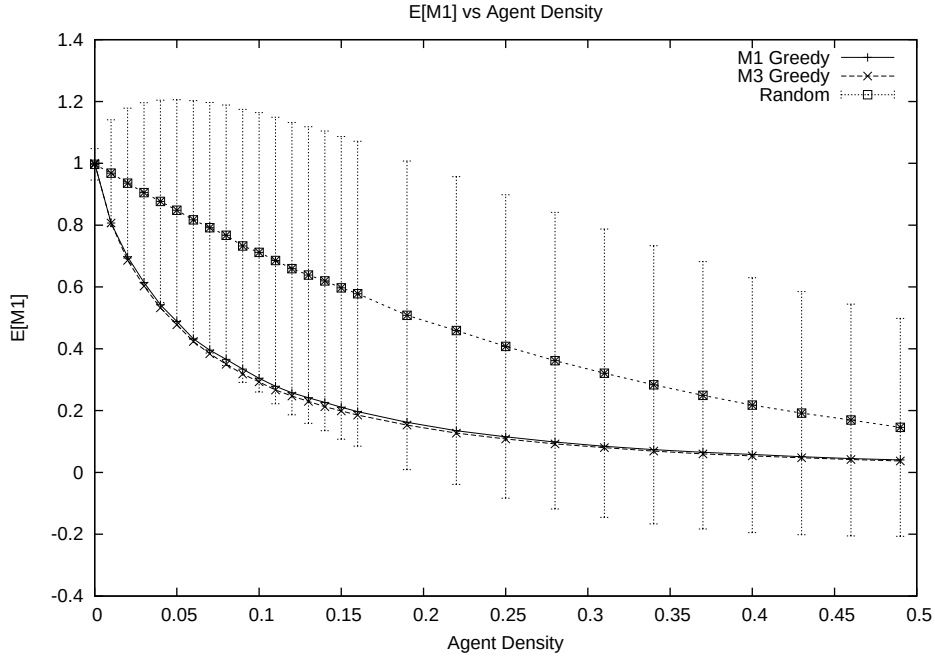


Figure 7.4.1: M1 versus agent density

experiment runs on the same Waxman [64] network of 200 nodes. The agent density ϵ was varied from 0.0 to 0.50 fraction of the links.

Results. Figure 7.4.1 shows E[M1] versus agent density curves for *Random*, *M1-Greedy*, and *M3-Greedy* algorithms. We see that M1-Greedy and M3-Greedy algorithms have similar performance with respect to E[M1] under identical agent deployment levels. It also shows that for smaller agent densities, the resulting values for M1 and M3-Greedy algorithms are one full standard deviation below the performance of the Random agent placement algorithm. At 10% percent agent deployment, the expected value of E[M1] for a random placement is 0.71, while M1-Greedy and M3-Greedy achieve E[M1] values of approximately 0.30. At this agent density, the difference between random and greedy algorithms is maximal. As agent density increases further, the performance of the three algorithms begin to coincide. This graph clearly shows that *M1* and *M2-Greedy* algorithms perform significantly better than *Random* placement of agents when we consider the E[M1]

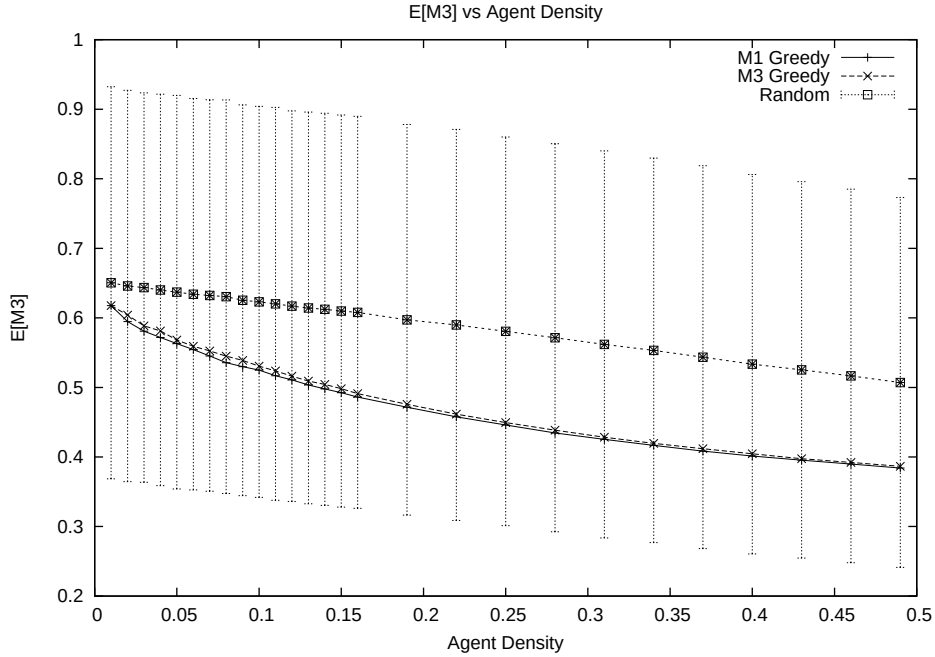


Figure 7.4.2: M3 versus agent density

values.

Figure 7.4.2 shows M3 versus agent density curves for *Random*, *M1-Greedy*, and *M3-Greedy* algorithms. Just as in Figure 7.4.1 the *M1-Greedy* and *M2-Greedy* outperform *Random* on the E[M3] measure. Curves for M1-Greedy and M3-Greedy decreases faster than the curve for *Random*. At 19% deployment the E[M3] value of Random is 0.59 while it is 0.471 and 0.475 for M1-greedy and M3-Greedy respectively. At this deployment level, the greedy algorithms performs approximately 20% better than Random. This performance advantage is maintained as deployment levels increase.

The results of Experiment 7.4.1 show that the M1 and M3-Greedy algorithms always perform better than Random. More surprisingly perhaps, they show that optimizing greedily with respect to M1 is in concordance with optimizing greedily with respect to M3. More precisely, a greedy placement of agents which sought to optimize E[M1] tends to be a

placement which is quite good with respect to E[M3] as well, and vice versa. The two new algorithms we have developed yield very effective agent deployments, even at low agent densities: A 10% deployment according to either of the proposed greedy algorithms can detect 70% of the attack flows (since $E[M1] \approx 0.3$) and trace back halfway to the attacking nodes (since $E[M3] \approx 0.5$).

7.4.2 Robustness of performance across networks of given size

Purpose. The purpose of the second set of experiments is to quantify the extent to which our conclusions in Experiment 7.4.1 might have been particular to the specific network in question.

Experiment setup. To explore this question, we considered the same experiment as in the previous section, but used a sequence of Waxman networks, each generated by the same process but with a different random number generator seed. Below we show the curves for just four of the networks, which were generated using random seeds 1, 2, 3, and 4 respectively.

Results. In Figure 7.4.3, the top row's two graphs consider the performance of the M1-Greedy algorithm on different Waxman networks, while the bottom row's two graphs consider the performance of the M3-Greedy. The two graphs in the left column consider the E[M1] measure, while the two graphs in the right column consider the E[M3] measure. As can be seen from the figures, the E[M3] measure is more sensitive to the choice of network (i.e. random seed) than the E[M1] measure. This is to be expected since E[M3] takes geometry and distance into consideration, where E[M1] is only concerned with geodesics (without reference to metrics). Also one can see that the graphs in the top row exhibit the

same sensitivity to the random seed, as the corresponding graphs in the bottom row. This reflects the fact that the M1-Greedy and M3-greedy algorithms produce solutions that are comparable with respect to both the E[M1] and E[M3] measures (as was noted in the first experiment).

The results indicate that conclusions drawn concerning the performance of the two schemes relative to each other are robust against the specific choice of network (of fixed size). Knowing this, we can now proceed to consider the impact of network size on the performance of the schemes.

7.4.3 Scalability of performance with respect to network size

Purpose. The purpose of the third set of experiments is to quantify the scalability of our conclusions with respect to ever larger networks.

Experiment setup. To explore this question, we carried out 8 experiments, which were identical except for the Waxman network size. Here, we report on the results of experiments on Waxman networks of sizes 100, 200, 400, and 800 nodes. By using this sequence of graphs, we determine the extent to which the Greedy algorithm's advantage (over random placement) is influenced by network size.

Results. In Figure 7.4.4, the top row's two graphs consider the performance of the M1-Greedy algorithm on different sized Waxman networks, while the bottom row's two graphs consider the performance of the M3-Greedy. The two graphs in the left column consider the E[M1] measure, while the two graphs in the right column consider the E[M3] measure. The top-left graph shows how the E[M1] curve changes when the M1-Greedy algorithm is used to place agents in different networks of different sizes. There are four curves in the

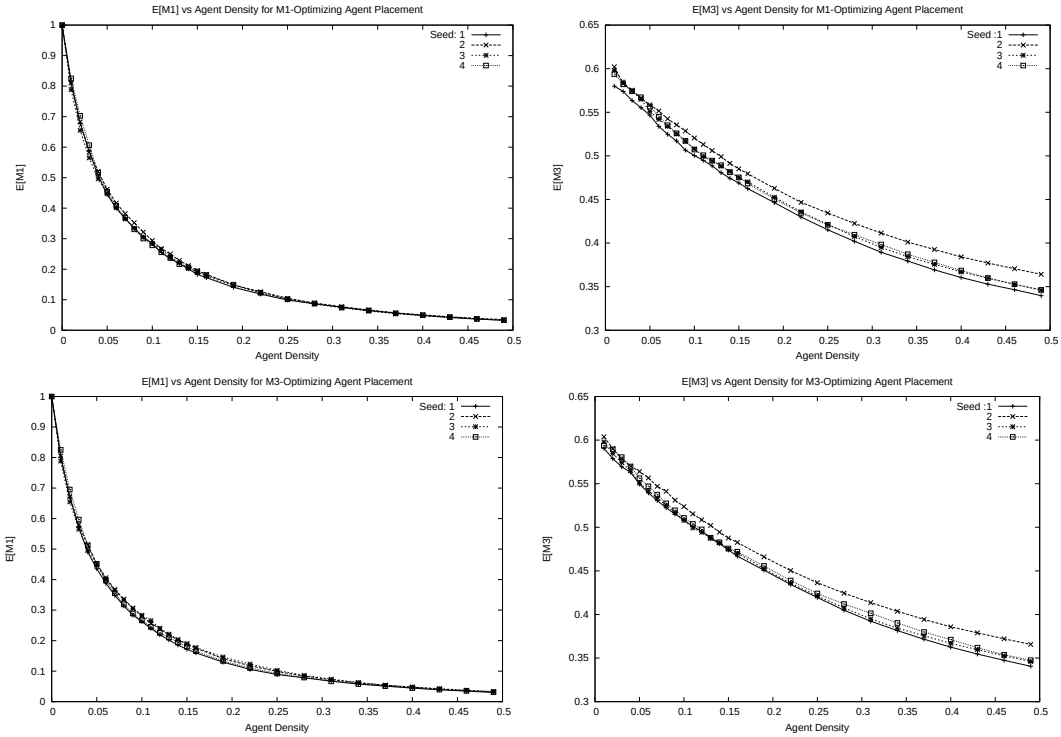


Figure 7.4.3: Greedy algorithm for varying networks of same size

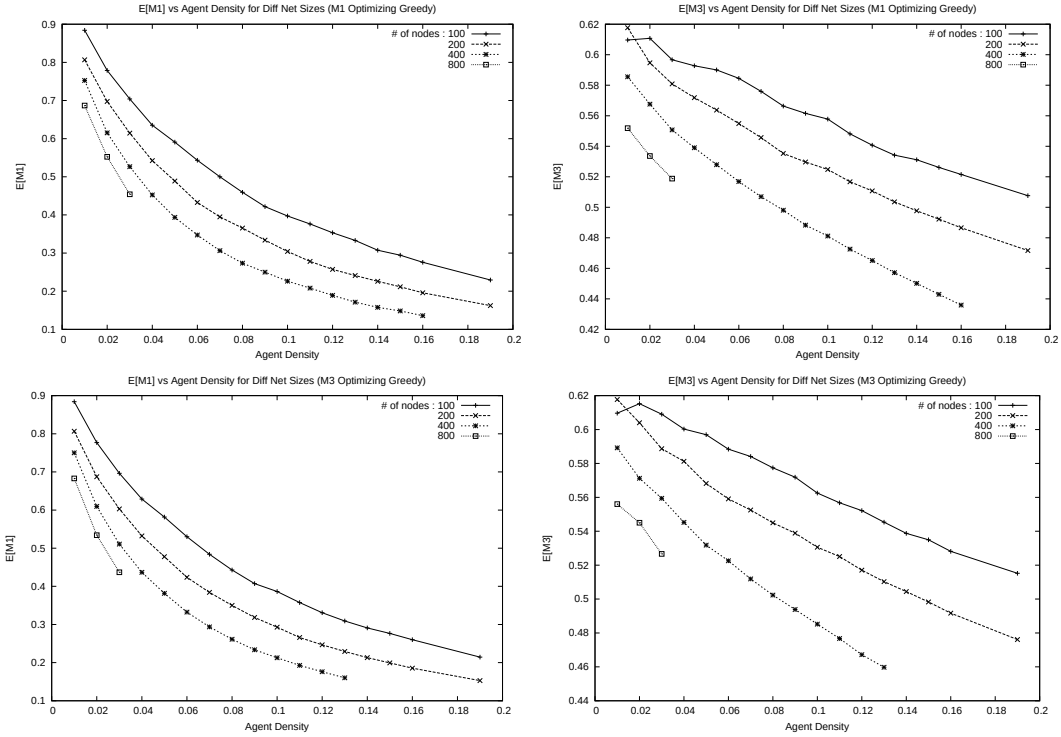


Figure 7.4.4: Greedy algorithm for varying networks sizes

graph, with each curve representing the results of the experiment for networks of a different size (100, 200, 400, and 800 respectively). Each of the curves individually shows similar characteristics. However, as the network size increases, we note that the entire curve shifts downward. At an agent density of 0.03 the E[M1] values for 100, 200, 400, and 800 node networks are 0.704, 0.614, 0.526, 0.454 respectively. As the number of nodes in the network doubles, the E[M1] value decreases approximately 13%.

The top-right shows how the E[M3] curve changes when the M1-Greedy algorithm is used to place agents in different networks of different sizes. Once again four curves in the graph represent the results of the experiment for networks of sizes 100, 200, 400, and 800 respectively. Each of the curves individually shows similar characteristics. Once again, as the network size increases, we note that the entire curve shifts downward.

The bottom-left figure is very similar to the top-left figure. It shows how the E[M1] curve changes when M3-Greedy algorithm is used to place agents in different networks of different sizes. For the agent density of 0.03 the E[M1] values for 100, 200, 400, and 800 node networks are 0.696, 0.603, 0.510, 0.436 respectively. As the number of nodes in the network doubles, the E[M1] value decreases approximately 14%. The graph shows us that the M3-Greedy agent placement performs better as the net size gets bigger.

The bottom-right is similar to the top-right figure. It shows how does the E[M3] curve behaves when M3-Greedy algorithm is used to place agents in different networks of different sizes. Once again the graph shows that as network size increases, the curve shifts downwards.

Taken together, the results of this experiment show that the performance of the proposed greedy algorithms *improves* for larger networks!

7.5 Refining greedy agent placement

In previous section we saw that the greedy algorithms provide better results compared to what is expected from random agent placements, *especially at low agent densities*. Greedy algorithms provides better results however the greedy results are not optimal.

Figure 7.5.1: Path network before agent placement

Consider M1-greedy placement of two agents in a path network as shown in figure 7.5.1. There are six nodes and five links in the network. Nodes are named as "v1, v2, v3, v4, v5, v6" respectively. M1-greedy algorithm will place the first agent on the link connecting nodes $v3$ and $v4$ as shown in figure 7.5.2.

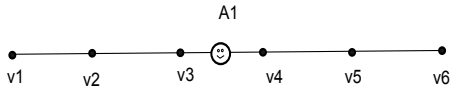


Figure 7.5.2: Path network after placement of first agent

When we run the algorithm for the first time to place the fist agent the link between $v3$ and $v4$ had the highest score so that the first agent had been placed on that link. After placing the first agent, all the remaining links had the same score as a result the second agent could have been placed on any of the remaining links. We chose to place it on the link between $v1$ and $v2$ as shown in figure 7.5.3.

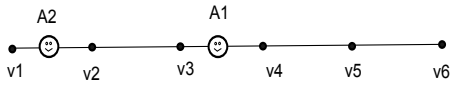


Figure 7.5.3: Path network after placement of second agent

After the placement of the second agent, the expected value of $M1$ became 0.27. However, the optimal value of $M1$ for this network with two agents should be 0.2 and one of the agents should have been placed between $v2$ and $v3$ and the other agent should have been

placed between $v4$ and $v5$ as shown in the figure 7.5.4. We have seen that the greedy

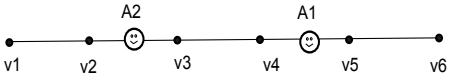


Figure 7.5.4: Path network after optimal agent placement

algorithms does not provide optimal solution. We now wonder if we can have better agent placement. Now we we will consider a sequence of increasingly sophisticated optimization algorithms. The first of these is called Iterative Greedy Agent Placement (IGAP).

7.5.1 Iterative Greedy Agent Placement (IGAP)

Iterative Greedy Agent Placement repeats the selected (M1-greedy or M3-greedy) algorithm for T iterations, numbered 0 to $T-1$. When $T = 1$, the greedy algorithm is run once in iteration 0, and the output is simply the output of M1-greedy or M3-greedy. When $T = 1$, the greedy algorithm is run twice. In the iteration 1, we remove each agent sequentially and add it again using the same greedy algorithm. When $T = i$, we start inductively from the placement at the end of iteration $i-1$, sequentially removing and re-place agents according to the greedy algorithm.

7.5.2 IGAP performance versus number of iterations

Purpose. The purpose of this experiment is to see how the see how does IGAP's performance influenced by the number of iterations for which the greedy algorithm is executed.

Experiment setup. For this experiment we used Waxman network of size 200, set agent deployment density to 10%, and used M1 and M3 greedy algorithms to place the agents. We collected the results after each iteration.

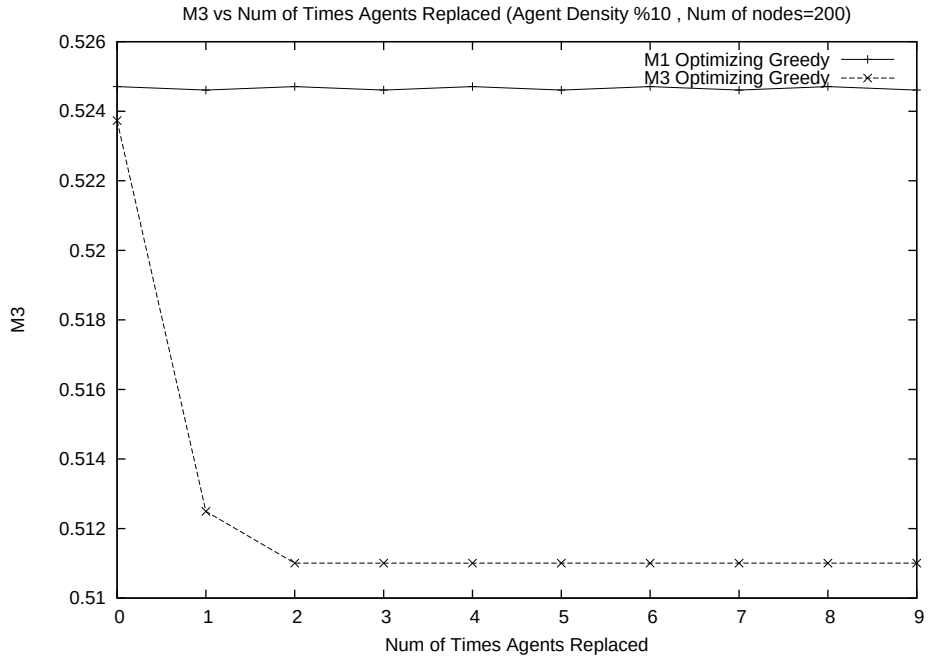


Figure 7.5.5: M3 versus number of agent replacements

Results. Figure 7.5.5 shows M3 versus number of iterations. There are two curves in the graph. The left one of them represents the results of M1-IGAP, and the right one represents the M3-IGAP. The y-axis shows M3 value and x-axis shows the number of iterations. At iteration 0 both curves has approximately the same value. As we move right on the x-axis the M3 curve drops 2% after the first iteration and drops 0.4% more after the second iteration. After this point M3 curve continues flat, which means the iterations after the 2^{nd} does not help at all. On the other hand, the M1 curve does not show any significant changes, which means iterations of M1-greedy algorithm does not help M3 values to get better.

Figure 7.5.6 shows two graphs. The figure on the left shows M1 versus the number of iterations and the figure on the right shows M2 versus the number of iterations. There are two curves in both graphs. One of them represents the results of M1-IGAP and the other one represents the M3-IGAP.

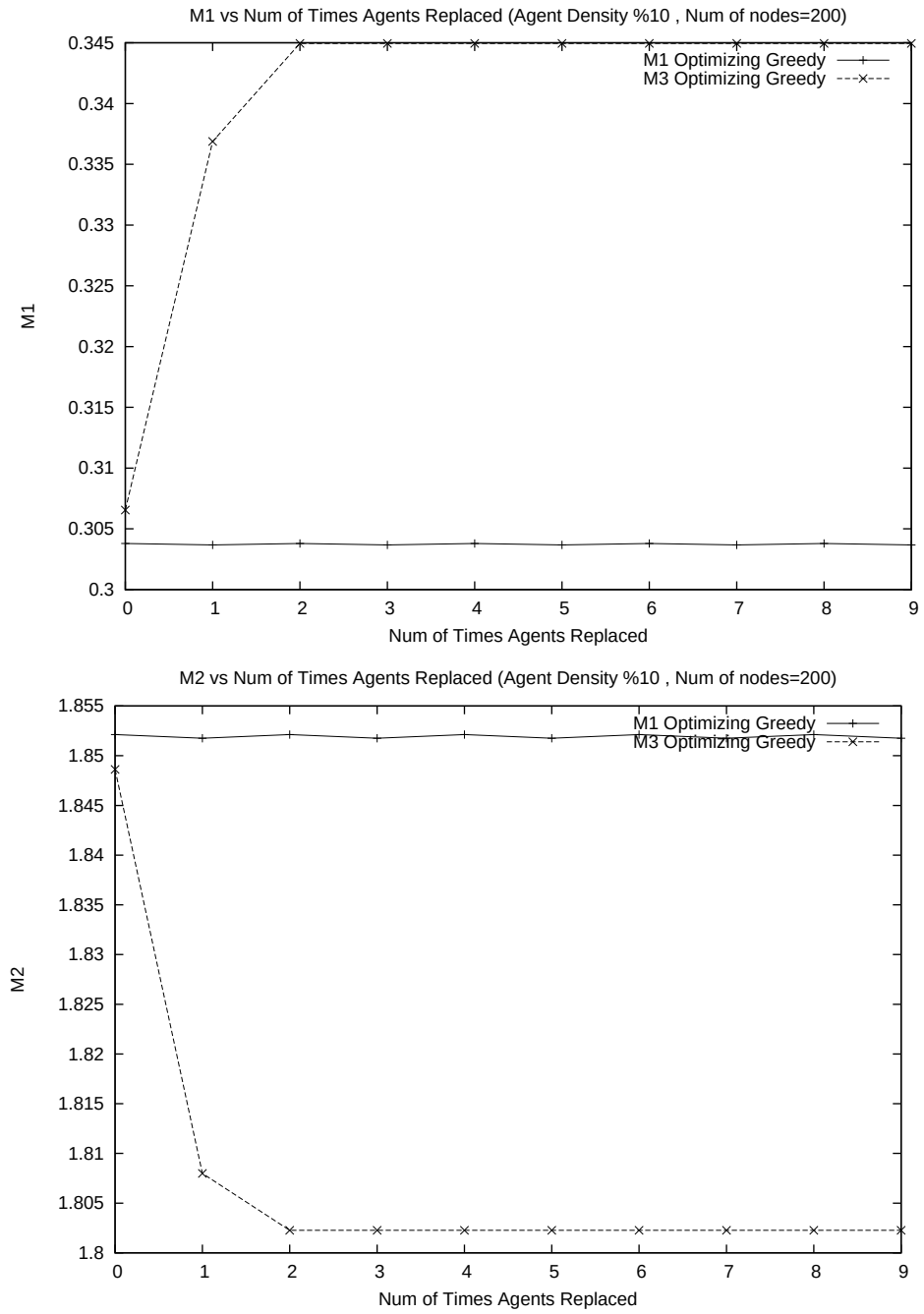


Figure 7.5.6: M1, M2 versus number of agent replacements

In the figure on the left, y-axis shows M1 value and x-axis shows the number of iterations. At iteration 0 both curves has approximately the same value. As we move right on the x-axis the M1 does show a horizontal movement, which means iterations of M1-greedy algorithm does help M1 result get better. On the other hand, the M3 curve increases 10% after the first iteration and increases 2% more after the second iteration. After this point M3 curve continues flat, which means the iterations after the 2nd does not help at all.

In the figure on the right, y-axis shows M2 value and x-axis shows the number of iterations. At iteration 0 both curves has approximately the same value. As we move right on the x-axis the M1 does again show a horizontal movement, which means iterations of M1-greedy algorithm does help M2 result get better. On the other hand, the M3 curve decreases 2% after the first iteration and decreases 0.4% more after the second iteration. After this point M3 curve continues flat, which means the iterations after the 2nd does not help at all.

In conclusion, results of experiment 7.5.2 showed that iterations of M1-greedy algorithm does not help any of the three performance measures (M1, M2, M3) get better. However, iterations of M3-greedy algorithm provides approximately 2.5% improvement for M2 and M3 values. On the other hand, M3-greedy decreases M1 results approximately by 13%. Recall that the M3 is 13% decrease in M1 value means that besides improving M3 value, the M3-greedy replaces agents such that new agent placement detect less attackers. This is likely to occur when the removed agent is the only agent that can detect attacks from some of the possible attackers. We saw that iterations of M1-greedy does not help improve our performance measures. Let's consider a path network with ten nodes and four agents. Nodes are named "v1, v2, v3, v4, v5, v6 ,v7, v8, v9, and v10" and agents are named "A1, A2, A3, A4" where the numbers represents the order the current agents were placed in the network. M1-greedy algorithm initially places the first agent on the link between $v5$ and $v6$, the second agent on the link between $v2$ and $v3$, the third agent on the link between

$v7$ and $v8$, and the fourth agent on the link between $v3$ and $v4$ as shown in figure 7.5.7.

Figure 7.5.7: Path network after M1-greedy agent placement

After placing agents in the network, for the first iteration, the IGAP algorithm removes the agent that was placed the first as shown in the figure 7.5.8.

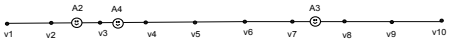


Figure 7.5.8: Path network with M1-greedy agent placement after first agent is removed

After removing the agent the IGAP algorithm finds the best place to place the last agent as the link from which it has just removed $A1$. After placing the new agent network topology does not change except the order of the agents as shown in figure 7.5.9.

Figure 7.5.9: Path network after the first iteration of M1-greedy agent placement

After placing agents in the network, for the second iteration, the IGAP algorithm removes the agent located in the link between $v2$ and $v3$ which was placed the first relative to the other agents as shown in the figure 7.5.10.

After removing the agent the IGAP algorithm finds the best place to place the last agent as the link between $v1$ and $v2$. After placing the new agent network topology does change as shown in figure 7.5.11 but the agent placement is not optimal yet.

After the second iteration, IGAP with M1-greedy do replace the agents on the same link where the previous agent has been removed from. Consequently, IGAP with M1-greedy does not help us find the optimal agent placement for the agents.

Figure 7.5.10: Path network with first iteration of M1-greedy agent placement after first agent is removed

Figure 7.5.11: Path network after the second iteration of M1-greedy agent placement

7.5.3 k-fold Iterative Greedy Agent Placement (IGAP-k)

In the previous experiment we saw how initial IGAP algorithm behaves. Recall that in the algorithm, for each iteration $i > 0$ we remove one already-placed agent, and then add the agent back again using a greedy algorithm. In this case, since there are already many agents placed, the number of remaining locations is limited, implying that the agent may be forced to return back to the spot from which it was removed. In order to prevent this, we modified $IGAP$ algorithm such that at each iteration $i > 0$ instead of removing one already placed agents, we remove k of them at once and than replace them again one-by-one using a greedy algorithm.

Purpose. The purpose of this experiment is to see the effects of k on the performance of IGAP-k, with respect to metric M3.

Experiment setup. For this experiment we used Waxman network of size 200, set agent deployment density to 10%, and used M3-greedy algorithms to place the agents. We collected the results after each iteration. We repeated the experiments for each $k = 1, 2, 4, 8, 16, 32$.

Results. Figure 7.5.12 shows the M3 versus number of iterations for different values of k. There are six curves in the graph, each curve represents results for different k values. The figure shows that each curve does a sharp decrease and after second iterations they all get horizontal and continue parallel to each other. After the second iteration they became

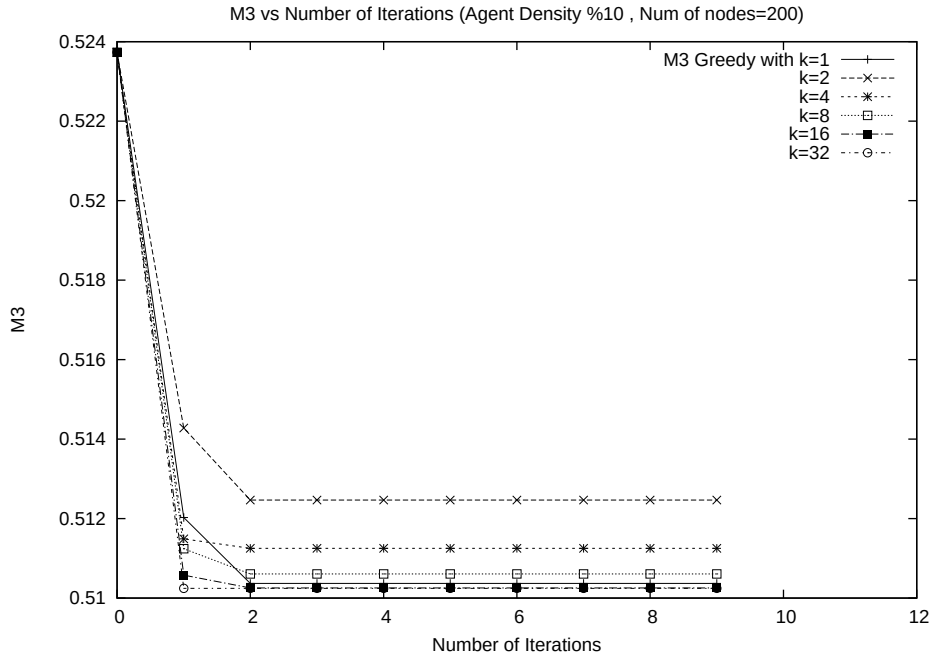


Figure 7.5.12: M3 versus number of agent replacements

perfectly flat. After the second iteration, as k increases from 1 to 2 the M3 value gets its worst value. For other values of $k > 2$, M3 values get better compared to M3 value for $k = 2$. When $k \geq 16$, the resulting M3 value is slightly better than the initial $M3$ value when k=1. The graphs show that increasing k does not improve M3-IGAP performance for M3 measures; results get even worse for values of $1 < k \leq 16$.

Figure 7.5.13 shows two graphs. The figure on the left shows M1 versus the number of iterations and the figure on the right shows M2 versus the number of iterations. There are six curves in both graphs, each curve represents results for different k values. In the figure on the left, y-axis shows M1 value and x-axis shows the number of iterations. As we move right on the x-axis the M1 values increases from 0.306 over 0.338. Increase in M1 value is higher for $k = 1$ compared to $k = (2, 4, 8)$. Since the lower the M1 value the better the results are, results again show that the M3-IGAP does not help M1 values to get better.

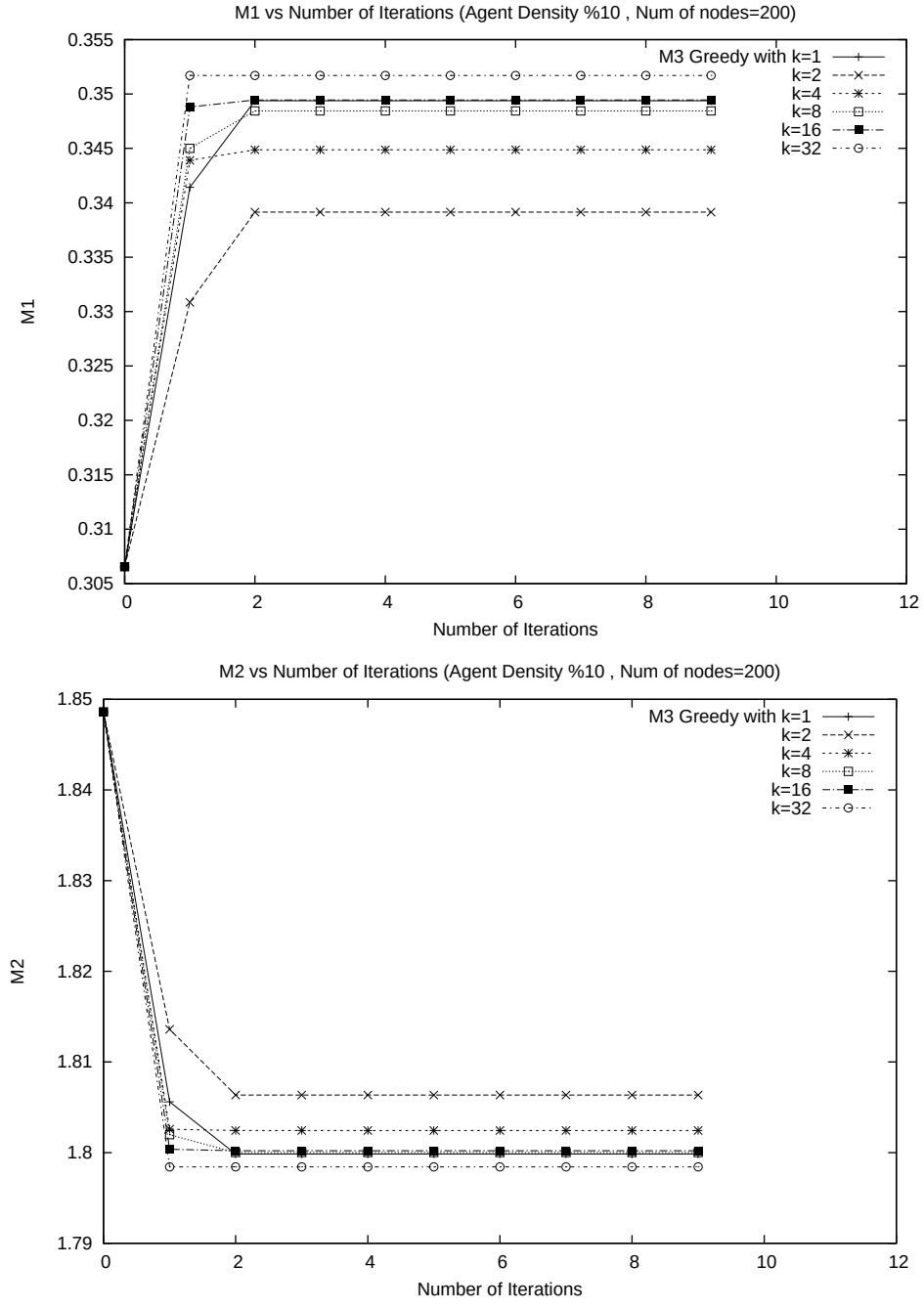


Figure 7.5.13: M1, M2 versus number of agent replacements

In the figure on the right, y-axis shows M2 value and x-axis shows the number of iterations. The curves behave in one to one correspondence to the curves in the the figure 7.5.12. Consequently results, show that removing k agents does not improve M3-IGAP results for M2 measures; results even get worse for values of $1 < k \leq 16$.

These experiments demonstrate that removing k agents from the top of the agents list does not help improve M3-IGAP performance. Moreover, at the point when the maximum value of k equals to the number of agents in the system, the performance results will be equal to the results when $k = 0$.

7.5.4 Randomized IGAP-k

In the previous experiment we saw that IGAP-k derives minimal benefits from larger values of k. Here we seek to determine the extent to which the way we select agents for replacement influences those results. The randomized IGAP-k algorithm differs from its predecessor in that it randomly permutes the order of the agents at the beginning of each iteration. As a consequence, a different random sequence of size k subsets are removed (and re-placed) in each iteration of the algorithm.

Purpose. To determine the effect of randomization on the IGAP-k algorithm's performance.

Experiment setup. For this experiment we used Waxman network of size 200, set agent deployment density to 10%, and used M3 greedy algorithms to place the agents. We collected the results after each iteration. We repeated the experiments for each $k = 1, 2, 4, 8, 16, 32$. For each k we first removed k agents randomly and the replaced k agents one by one.

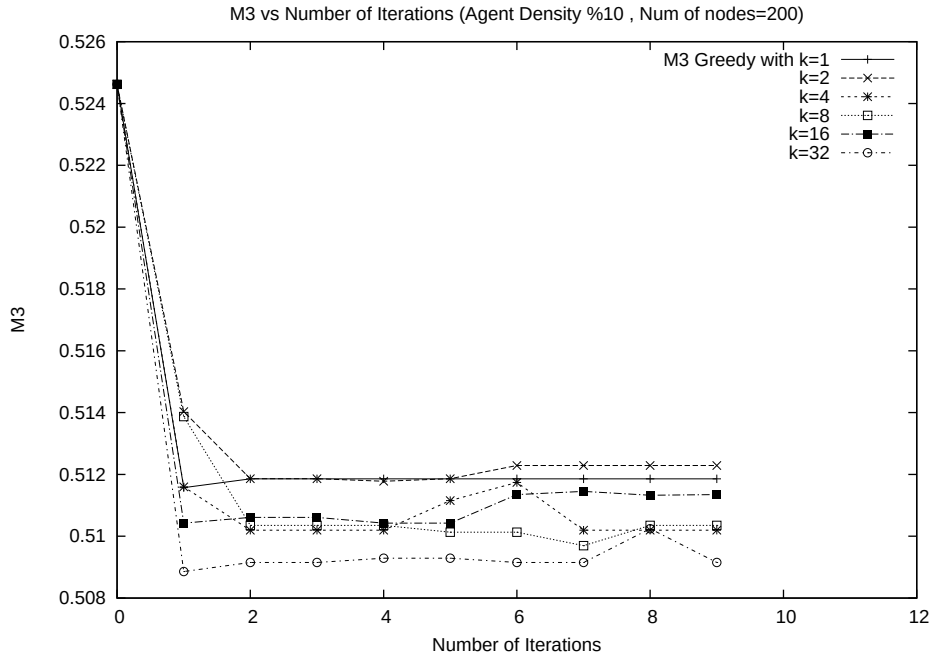


Figure 7.5.14: M3 versus number of agent replacements with error bars

Results. Figure 7.5.14 shows the M3 versus number of iterations for different values of k. There are six curves in the graph, each curve represents results for different k values. We see that as we move from iteration 0 to iteration 1 all of the curves makes a sharp decrease similar to the previous results. After that point three of the lines decreases some more and rest of the lines increases slightly. After the second iteration all the lines performs at least as good as the line for $k = 1$. At this point $k = 32$ performs 0.1% better than $k = 1$. After this point, unlike the previous experiments, curves do not follow a straight flat line, instead they go up and down keeping while keeping the same average horizontal movement along the x-axis. However the M3 values does not get better than it was after second iteration. Consequently, we can say that doing two iterations with random agent removal helps M3 value get better.

Figure 7.5.15 shows the results for M1 and M2 values. As seen from the figures, randomized M3-IGAP-k with $k > 1$ does not help M1 score improve significantly. In Figure 7.5.15 the

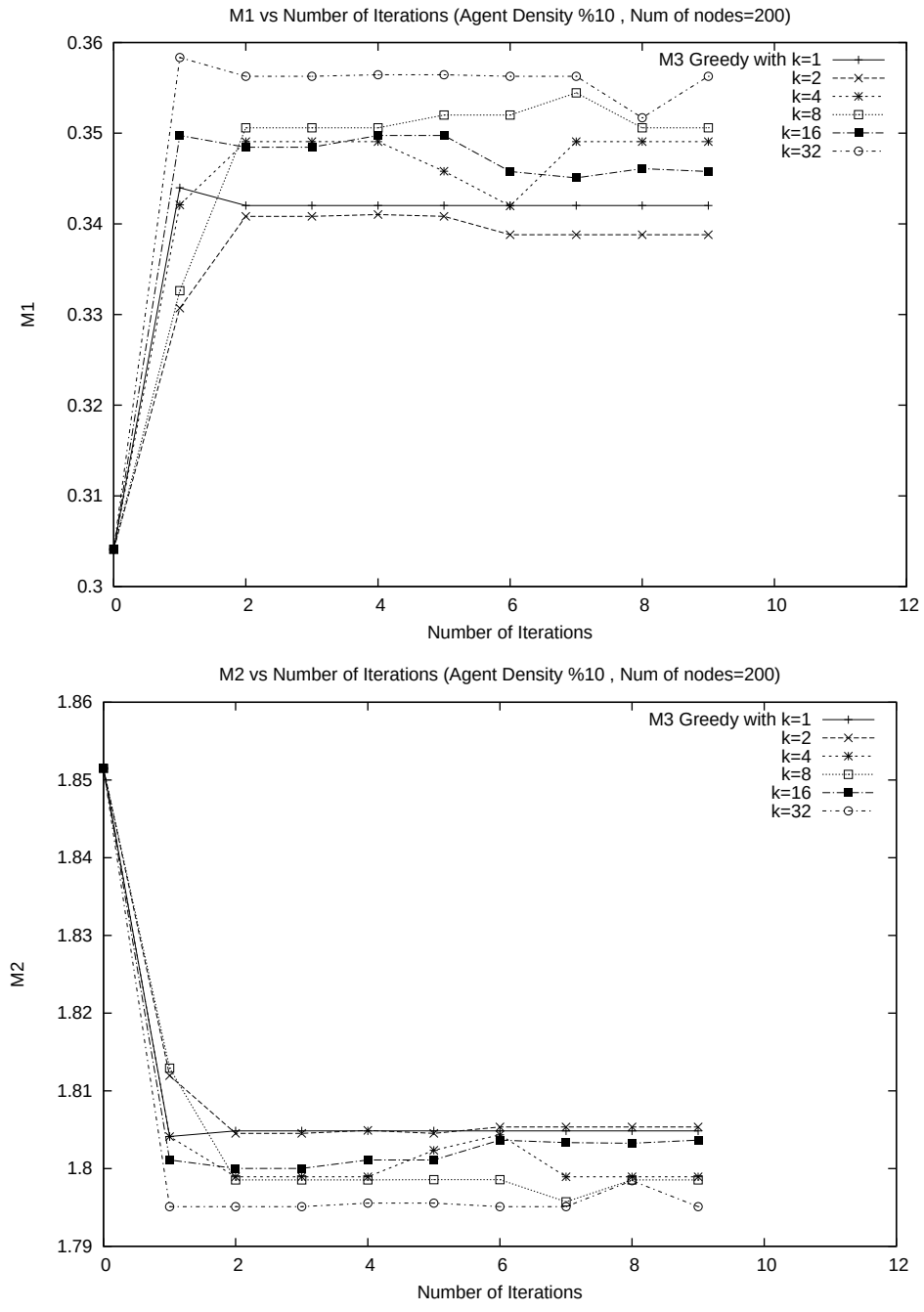


Figure 7.5.15: M1, M2 versus number of agent replacements

graph on the right shows the results for M2 values. The M2 values behave similar to their M3 analogues.

In section 7.5 we have explored the effects of repeating greedy algorithms on the performance of the agent system. We have seen that the iteration of M1-greedy algorithm does not help at all. On the other hand iteration of M3-greedy algorithm (with or without the k-fold option and randomization) improves M2 and M3 performance of the system. These studies show that randomized k-fold M3-greedy may be a promising bootstrapping of the M3-greedy algorithm.

CHAPTER 8

EXTENSIONS: ATTACKER LOCALIZATION

We saw in previous chapters that the proposed agent system is capable of reconstructing attack flows even when deployment levels are modest. In addition, we showed that system performance can be improved by careful placement of agents within the network. All of our efforts to date made use of information about a *single* attack. Now we seek to go further.

We start from the observation that a malicious attacker is unlikely to instrument a Botnet merely to execute a single attack. This is because the construction of a Botnet requires significant investment of time, resources, which requires being for the criminal elements to be subject to considerable risk. Once established, the Botnet represents a capital investment from which revenue can be derived. Thus, it is very likely that after a Botnet has been established, it will be used to execute *multiple attacks over time*. The question naturally arises: Is there some way in which the agents in our system can aggregate information collected over longer timescales in order to further isolate candidate locations of malicious nodes? In this chapter we demonstrate initial results that indicate an affirmative answer to this question.

In developing our approach, we will assume a continuity in the composition of the set of attacking nodes. That is to say, we will assume that the same set of attackers orchestrate a sequence of attacks on different victims, over time. We will not consider changes in membership of the set of attackers, nor will be consider changes to network topology, or

changes to the routing table. This idealized setting will allow us to evaluate the effectiveness of proposed schemes at aggregating attack structure information over long time scales. In subsequent work (outside the scope of this thesis) we will quantify the sensitivity of our approach to the the previously stated assumptions.

8.1 Attacker Localization Problem

We define attacker localization problem (ALP) as follows.

- Input: A set of flow reconstruction problems (FRPs) $\mathcal{P} = (P_1, P_2, ..., P_n)$ and set of corresponding flow reconstruction problem solutions $\mathcal{L} = (L_1, L_2, , L_n)$, where each FRP instance $P_i = (G, R, A, D, v_i)$ differs only in the victim identity v_i, each $L_i = (S_i, E_{S_i})$ is a maximal valid solution to P_i, and $|D| > 0$.

Given an instance of the attacker localization problem, we define a solution as:

- Output: A map from $s : V[G] \to \mathbb{R}$ which assigns a "Suspicion level" to each vertex in V, and a threshold $\tau \in \mathbb{R}$.

We interpret this output as asserting that the set of vertices

$$\mathcal{S} = \{v \in V \mid s(v) > \tau\}$$

may be justifiably treated as *suspects* in the attack sequence $\mathcal{P}$. It will also be useful to consider a parametrically defined subset of the suspects:

$$\mathcal{S}(t) = \{v \in V \mid s(v) > t\}.$$

Given this formal structure, how might we assess a solution (s, τ) to an instance of ALP $(\mathcal{P}, \mathcal{L})$? Two natural measures come to mind:

- False positive rate (FPR) : The number of suspects who are innocent (as a fraction of the total number of suspects). Stated precisely, this quantity is

$$FPR((\mathcal{P}, \mathcal{L}), (s, \tau)) = \frac{|\{v \in V/D | s(v) \geqslant \tau\}|}{|\{v \in V | s(v) \geqslant \tau\}| + \epsilon},$$

  where $\epsilon > 0$ is any small positive quantity.

- False negative rate (FNR) : The number of guilty parties (attackers) who are not suspected, as a fraction of the total number of attackers. Stated precisely, this is

$$FNR((\mathcal{P}, \mathcal{L}), (s, \tau)) = \frac{|\{v \in D | s(v) < \tau\}|}{|D|}$$

For brevity, in what follows, whenever the problem instance $(\mathcal{P}, \mathcal{L})$ and solution (s, τ) are clear from the context, we will write FPR in place of $FPR((\mathcal{P}, \mathcal{L}), (s, \tau))$ and FNR in place of $FNR((\mathcal{P}, \mathcal{L}), (s, \tau))$.

We expect a natural trade off between the two measures above, since it is easy to achieve either very low FPR or very low FNR. To see this, let s be the function which assigns all vertices in V a score of 1. To achieve an FPR=0, it suffices to take $\tau = 2$. To achieve an FNR=0, it suffices to take $\tau = 0$. In general, however, the problem of achieving low FPR and FNR simultaneously is likely to be very difficult.

8.2 Proposed solution

We present the design of a new algorithm, which we call the "Searchlight-based Localization Algorithm for Network Tomography (SLANT)". The SLANT algorithm seeks to locate sets of attackers who collaborate repeatedly to attack different victims over time–that is to say, it solves instances of the attacker localization problem. We will describe it first as a centralized implementation. Later, we will exposit a distributed version. To begin, however, some preliminary definitions are required:

Definition 8.2.1. *Given a graph $G = (V, E)$ and a consistent, symmetric routing table R. Let $a, v \in V$ be any two distinct vertices. We define the* **cone** *$C(a, v)$ as*

$$C(a, v) = \{u \in V | a \in F(u, v)).$$

Informally stated, the cone $C(a, v)$ consists of those vertices u for which the flow from u to v (with respect to routing table R) passes through a. In the case where a is an agent and v is a victim, the cone $C(a, v)$ represents a set of candidate locations for attackers.

Given an instance ALP, consider the ith constituent instance of FRP $P_i = (G, R, A, D, v_i)$ within it, together with the provided solution $L_i = (S_i, E_{S_i})$. Each agent $a \in S_i$ produces a cone $C(a, v_i)$. Several questions arise:

1. How should the cones $\{C(a, v_i) \mid a \in S_i\}$ be combined to produce the suspicion function s in response to attack i?

2. How should changes in the suspicion function in response to attacks be accumulated, over the entire sequence $(i = 1, \ldots, n)$ of attacks?

3. How should the threshold of suspicion τ be selected?

Rather than committing to any single answer, we provide a general framework for answering these questions:

1. The cones $\{C(a, v_i) \mid a \in S_i\}$ obtained with reference to agents during attack i are to be combined defining

$$\Delta\tau(u, i) = \sum_{a \in S_i} \sigma(u, C(a, v_i), S_i),$$

   for each $u \in V$, where σ is the **multi-agent information combining function**. Several choices of σ are proposed below, of which a subset are considered in this research, while others will be considered in future work.

2. Changes in the suspicion function $\Delta\tau(u, i)$ are accumulated over the entire sequence $(i = 1, \ldots, n)$ by defining

$$\tau(u) = \sum_{i=1}^{n} \rho(\Delta\tau(u, i), i, n),$$

   where ρ is the **multi-attack information combining function**. Several choices of ρ are proposed below, of which a subset are considered in this research, while others will be considered in future work.

There are several possible natural choices for the multi-agent information combining function and multi-attack information combining function. Here we group our choices into three natural classes:

- **Union set score assignment** (USSA): Here we take ρ to be

$$\rho(x, i, n) = \begin{cases} 0 & \text{if } x < 1 \\ 1 & \text{otherwise.} \end{cases}$$

and σ to be

$$\sigma(u, C, S) = \begin{cases} 1 & \text{if } u \in C \\ 0 & \text{otherwise.} \end{cases}$$

This scheme considers each separate attack additively. For each attack, it takes the union of all agent cones and increments the suspicion score of vertices within the unioned set. This means that for each attack in the series, a vertex can accumulate at most 1 towards its suspicion score.

- **Pure additive score assignment** (PASA): Here we take ρ to be

$$\rho(x, i, n) = x$$

and take σ as previously defined for USSA:

$$\sigma(u, C, S) = \begin{cases} 1 & \text{if } u \in C \\ 0 & \text{otherwise.} \end{cases}$$

This scheme considers each separate attack additively. For each attack, it the increments each vertex's suspicion score by the number of agent cones in which the vertex lies. This means that for the ith attack in the series, a vertex can accumulate at most $|S_i|$ towards its suspicion score.

- **Normalized additive score assignment** (NASA): Here we take ρ as

$$\rho(x, i, n) \quad = \quad x/n$$

and define σ as:

$$\sigma(u, C, S) \quad = \quad \begin{cases} 1/|S| & \text{if } u \in C \\ 0 & \text{otherwise.} \end{cases}$$

This scheme considers each separate attack additively. For each attack, each vertex accumulates a suspicion score equal to the *fraction* of agent cones in which the vertex lies, normalized by the total number of attacks n. Thus, a vertex achieves a suspicion score of 1 if and only if it is in every agent cone reported in every attack.

8.2.1 Example

We use an example to illustrate how $SLANT$ works. Consider the attack scenario shown in Figure 8.2.1. In this example scenario, we consider

- A grid network $G = (V, E)$ where $V = (v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, d)$.
- A routing table R.
- An agent set $A = (A_1, A_2, A_3, A_4)$.
- An attacker set $D = \{d\}$.

In the figure, dashed arrows represent routes from each vertex to the victim. Smiley faces represent agents.

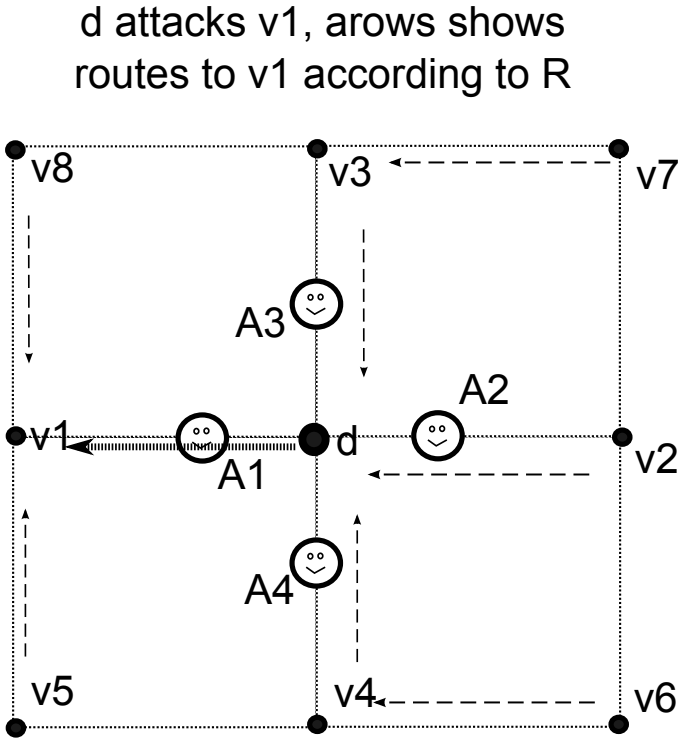


Figure 8.2.1: Attacker d attacks on victim v1

The attacker d starts by attacking victim v_1; a bold dashed arrow between d and v_1 represents the attack traffic flow. The attack traffic goes through A_1 which detects the attack. The solution to the flow reconstruction problem is determined to be $L = (A_1, \emptyset)$. Although A_1 doesn't precisely know where the attacker is, the cone $C(a_1, v)$ is $\{(v_2, v_3, v_4, v_6, v_7, d\}$. Since A_1 is the only agent in solution set L, the resulting set of suspicious vertices consists of just $C(a_1, v)$. If we assume the $USSA$ for score assignment then the process produces the following assignment of suspicion scores:

v	$\tau(v)$
v_1	0
v_2	1
v_3	1
v_4	1
v_5	0
v_6	1
v_7	1
v_8	0
d	1

If the suspicion threshold τ is taken as 0, then the suspects $\mathcal{S}(0)$ would be $\{v_2, v_3, v_4, v_6, v_7, d\}$. The FPR of this solution is 5/6 since 5 of the 6 suspects are innocent. The FNR of the solution is 0, since no guilty party has evaded inclusion in the suspicious set.

In Figure 8.2.2 nodes are colored according to their suspicion scores: Black, blue, green, yellow, and red colors represent suspicion scores 0, 1, 2, 3, and 4 respectively. We have illustrated the aforementioned solution to the ALP instance consisting of one attack by d on v_1.

In figure 8.2.3 there are four figures, each of which follows the same colorization rules: Black, blue, green, yellow, and red colors represent suspicion scores 0, 1, 2, 3, and 4 respectively. The shaded areas represent the agent cones.

The top left figure represents the result of $SLANT$ algorithm after one attack; the figure on top right represents the $SLANT$ results after two attacks, the figure on bottom left represents the $SLANT$ results after three attacks and the figure on bottom right represents

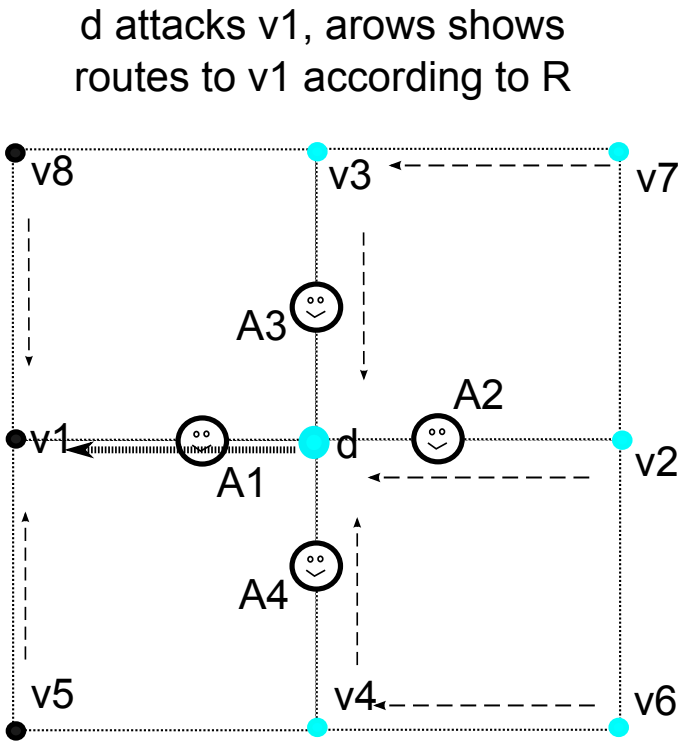


Figure 8.2.2: Possible attackers after d attacks on victim v1

the $SLANT$ results after four attacks.

After the second attack, if we continue to take the suspicion threshold τ as 0, then the suspects $\mathcal{S}(0)$ would be $\{v_2, v_3, v_4, v_6, v_7, d\}$, yielding an FPR of 7/8 and $FNR = 0$. If, however, we change the value of threshold to 1, then the corresponding $\mathcal{S}(1)$ becomes $\{v_2, v_4, v_6, d\}$, making FPR=$\frac{3}{4}$ and FNR=0. Since 3/4 is lower than 5/6 (the FPR after one attack), this shows that in this example, it is possible to combine information from multiple attacks to yield lower FPR metrics, while preserving low FNR metrics.

After the third attack $|\mathcal{S}(0)| = 9$ and FPR= $\frac{8}{9}$. On the other hand, if the threshold τ is taken as 1, then $|\mathcal{S}(1)| = 6$ and FPR= $\frac{5}{6}$. If the threshold τ is taken as 2, then $|\mathcal{S}(2)| = 2$ and FPR= $\frac{1}{2}$. Once again, note that 1/2 is lower than 3/4, which was the FPR attained after 2 attacks, thus information from the third attack can be incorporated and used to localize the attacker further.

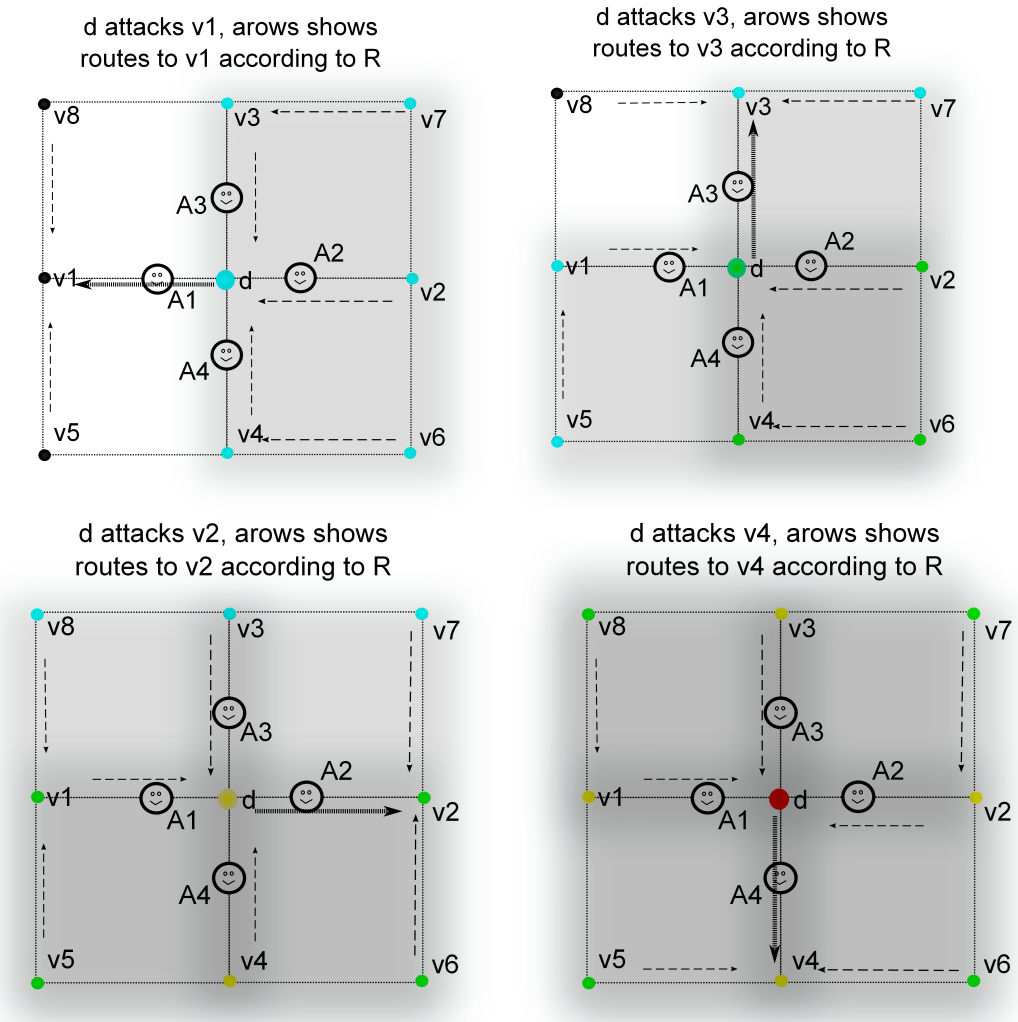


Figure 8.2.3: Possible attackers after four attacks

Finally, after the fourth attack, the value of $|\mathcal{S}(0)| = 9$ and FPR= $\frac{8}{9}$; with a threshold τ of 1 we get $|\mathcal{S}(0)| = 9$; with a threshold τ of 2 we get $|\mathcal{S}(2)| = 5$. Finally, with a threshold τ of 3, we get that $\mathcal{S} = \{d\}$. FPR=FNR=0. Now the attacker has been completely localized!

The previous example shows that it is possible to combine information from multiple attacks to localize the attacker location, but the choice of threshold τ is critical to the reduction of FPR. We give a brief formal description of how the SLANT algorithm computes the suspicion function s below. Then, in what follows, we will investigate the impact of the choice of value of the suspicion threshold parameter τ on the FPR and FNR metrics, in greater detail.

8.2.2 Centralized SLANT Algorithm

The centralized implementation of the $SLANT$ algorithm operates as follows.

Given as input, an ALP instance, that is a set of flow reconstruction problems $\mathcal{P} = (P_1, P_2, ..., P_n)$ and corresponding solutions $\mathcal{L} = (L_1, L_2, , , L_n)$, where each FRP instance $P_i = (G, R, A, D, v_i)$ differs only in the victim identity v_i, each $L_i = (S_i, E_{S_i})$ is a maximal valid solution to P_i, and $|D| > 0$.

- Define $s(v) = 0$ for all v in V.
- For each i from 1 to n:
  - For each a in S_i:
    - Compute the cone $C(a, v_i)$.
  - For each vertex $v \in V$:

* Compute $\Delta\tau(u,i) = \sum_{a \in S_i} \sigma(u, C(a, v_i), S_i)$, using the appropriate σ based on the scheme being used.

– Increment $s(v)$ by $\rho(\Delta\tau(u,i))$.

8.3 Experimental Setup

In order to determine the effectiveness of the *SLANT* algorithm, we conducted many simulation experiments. Two different sets of experiments were carried out:

- First, we wanted to do preliminary experiments to verify SLANT's operation (and compare it with out intuitive expectations). This was most easily done in a grid network using only *one* attacker.
- Next we wanted to see how SLANT performed in more general artificial networks of Waxman type when more than one attacker is present.

8.3.1 Grid networks

For the experiments with grid-like network the experiment setup is as follows.

- We begin by taking a geometric square of side $\lfloor\sqrt{|V|}\rfloor$. Within this we place $\lfloor\sqrt{|V|}\rfloor^2$ nodes, at each lattice coordinate (i,j) where $i,j \in \{0,\ldots,\sqrt{n}-1\}$. This establishes the vertex set V. Then we created horizontal and vertical links between pairs of nodes whose x coordinates differ by 1 (but y coordinates are identical), or whose y coordinates differ by 1 (but x coordinates are identical). This establishes the edge

set E. In this way, we have constructed a graph $G = (V, E)$ which has the form of a Cartesian grid.

- We instantiate a routing table for this network by running Dijkstra's algorithm from all nodes.
- After creating the vertices and the links we build the agent set A by placing an agent along every k^{th} horizontal and every k^{th} vertical link, resulting in an agent density of $1/k$. For example, when $k = 2$, the resulting agent set A has a density of 0.5.
- After that we randomly select the attackers, forming the set D. In our experiments, $|D| = 1$ or $|D| = 10$.
- We then chose a sequence of n victims $v_1, \ldots v_n$, uniformly at random. In the experiments, n varies from 10 to 80.

A grid experiment is thus completely characterized by $|V|$, $|k|$, $|D|$ and n.

Once the experiment is specified, the simulation operates as follows: The attackers D sequentially attack the specified random sequence of victims $v_1, \ldots v_n$, yielding n FRP instances. The agents A operate to produce n maximal valid solutions to these FRP problems. Together, the n problems and solutions constitute an instance of the attacker localization problem. This is given to the SLANT algorithm, to produce a suspicion map s, for the vertex set V. Given the suspicion map s, one can ask how the choice of normalized threshold NT, affects the size of the suspicious set SS where

$$SS = \{v \in V \mid s(v) > NT\},$$

and how the choice of NT influences the FPR

$$FPR = \frac{|\{v \in V - D | s(v) > NT\}|}{|\{v \in V | s(v) > NT\}| + \epsilon}.$$

The two variables we consider in our experiments are $|SS|$ and FPR; the parameters we vary are k, n, $|V|$.

8.3.2 Waxman networks

For the experiments with Waxman network the experiment setup is as follows.

- We begin by taking a geometric square of side $5\sqrt{|V|}$. Within this we place $|V|$ nodes uniformly at random. This establishes the vertex set V. Then we add random links between nodes, chosen with probability that is inversely proportional (exponentially) to the Euclidean distance of two nodes. This link addition process continues until the desired edge density is reached. Once the desired edge density is reached, we check if the network is connected or not. If it is we return the network. Otherwise, we continue to add edges until the graph becomes connected. In our experiments, we took the edge density to be slightly above two (2.1); this value was chosen because it is close to the value obtained through analysis of the present Internet topology [68]. This establishes the edge set E. In this way, we have constructed a graph $G = (V, E)$ which has the form of a Waxman network whose edge density reflects present edge densities in the Internet.

- We instantiate a routing table for this network by running Dijkstra's algorithm from all nodes.

- After creating the vertices and the links we build the agent set A by placing an agent along $\epsilon|E|$ edges, chosen use the M3-greedy agent placement algorithm which is described in section 7.3.

- After that we randomly select the attackers, forming the set D. In our experiments, $|D| = 1$ or $|D| = 10$.

- We then chose a sequence of n victims $v_1, \ldots v_n$, uniformly at random. In the experiments, n varies from 10 to 80.

A Waxman experiment is thus completely characterized by $|V|$, ϵ, $|D|$ and n.

Once the experiment is specified, the simulation operates as follows: The attackers D sequentially attack the specified random sequence of victims $v_1, \ldots v_n$, yielding n FRP instances. The agents A operate to produce n maximal valid solutions to these FRP problems. Together, the n problems and solutions constitute an instance of the attacker localization problem. This is given to the SLANT algorithm, to produce a suspicion map s, for the vertex set V. Given the suspicion map s, one can ask how the choice of normalized threshold NT, affects the size of the suspicious set SS where

$$SS = \{v \in V \mid s(v) > NT\},$$

and how the choice of NT influences the FPR

$$FPR = \frac{|\{v \in V - D | s(v) > NT\}|}{|\{v \in V | s(v) > NT\}| + \epsilon}.$$

The two variables we consider in our Waxman experiments are $|SS|$ and FPR; the parameters we vary are $|V|$ and $|D|$.

8.4 Experiments

The example in the previous section shows that SLANT is able to combine information from multiple attacks to localize the attacker, but the choice of threshold τ is critical to the reduction of FPR. Although we conducted experiments with USSA and PASA schemes, we found the performance of these earlier schemes (with respect to FPR and FNR) to be significantly worse than what was achieved by the NASA scheme. For succinctness, in what follows, we report only on the experiments where the NASA scheme was used. Specifically, we investigate the impact of the choice of value of the suspicion threshold parameter τ on the FPR and FNR metrics, in greater detail. Throughout many of the experiments that follow, the independent variable that we consider is referred to as the Normalized Threshold (NT), by which we mean the value of τ used to define the suspicious set (SS) with respect to the NASA scheme. We seek to find the relationship between NT and FPR and the size of the SS.

8.4.1 Visualizing $SLANT$ localization convergence

Purpose. The purpose of this experiment is to see how the suspicion scores changes as the number of attacks n progresses. We sought to visualize the results and compare the results with the example in the previous section.

We use a 100 node grid topology, fixing agent density at 50%, number of attackers D to be 1, and considering the state of the suspicion map s after varying numbers of attacks $n = 1, 2, 4, 6, 8, 10, 100$. The attacker is located at coordinates (7,2). The sequence of victims attacked were chosen uniformly at random.

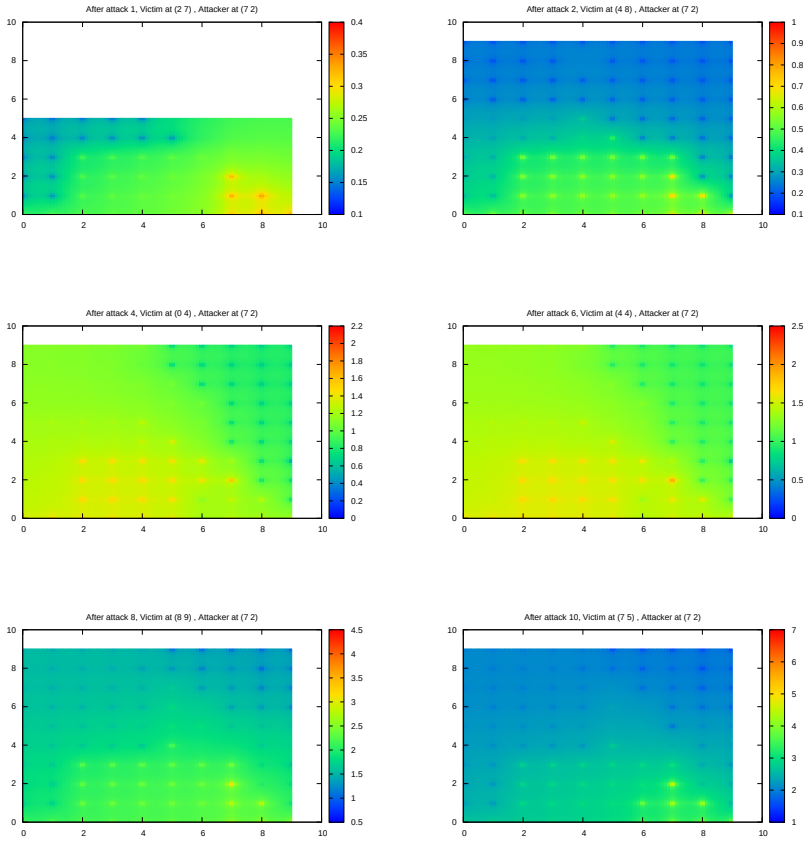
Results. Figure 8.4.1 shows the suspicion scores of each vertex after the aforementioned number of attacks. The color distribution represents the suspicion scores, *normalized against the maximum suspicion score of all vertices in the network.*

The top left graph shows the scores after the first attack. We see three vertices as the most suspicious ones, and vertices above $y = 5$ are not suspected at all, since they have score 0. In the top right figure we see that the previous three vertices still have the highest scores. In addition, the size of the suspicious vertices set increases. After the fourth attack, vertices on the right side of the figure get higher suspicion than before, though at the same time the real attacker has the highest score. After the sixth attack again the real attacker has the highest score, however, there are several other nearby vertices who have high scores too. After the tenth attack the difference between the scores gets more pronounced, and only the three vertices closest to the true attacker have comparable high scores, while the rest of the network vertices have very small scores. However, the attacker still has the highest score. After the tenth attack there is no significant change in the figure. This shows that for this case we can easily determine the attacker after tenth attack, as seen in the figure and that $SLANT$ is a promising algorithm for attacker localization.

8.4.2 Varying network size

Purpose. The purpose of this experiment is to see how $|SS|$ and FPR varying network size $|V|$ as $100, 1000$. We used grid networks, varying network size $|V|$ between 100 and 1000. We fixed $|D| = 1$, $n = 80$ attacks, δ=50%.

Results. Figure 8.4.2 shows NT vs. $|SS|$ and FPR. There are two graphs. The one on the top shows results for a network size of 100 nodes and the one on the bottom shows results



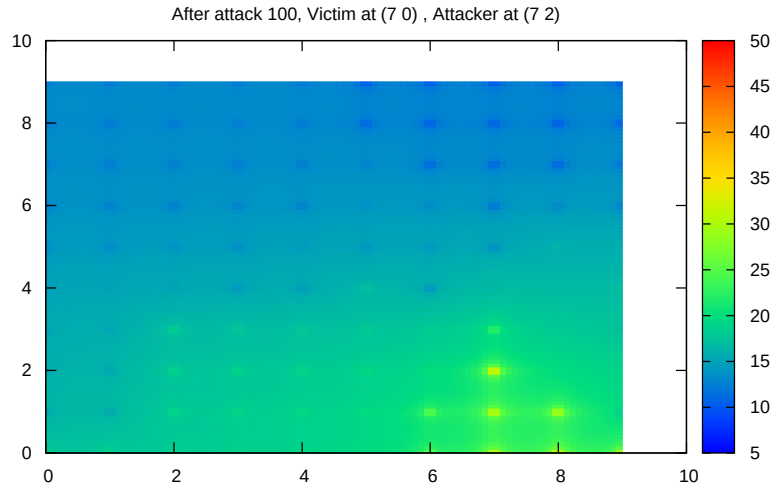


Figure 8.4.1: Localization after different number of attacks

of experiment for a network size of 1000 nodes. There are two curves in each graph one of them represent the number of vertices in the final suspicious set and the other represents the false positive ratio. For convenience, to show both curves on the same graph, we plot $10 * FPR$ and $|SS|$.

In the graph on the top, we see that both curves monotonically decrease. This result matches the phenomenon observed in the earlier example we gave to describe the $SLANT$ algorithm. The decrease in FPR shows that as the threshold value increased the number of vertices in the final suspicious vertices set decreases, but the attacker stays in the SS until all the vertices are removed from the set. This result supports our intuition that the attacker will have the highest suspicion score. If it was not true, then there would be increases in the FPR curve. The $|SS|$ curve also monotonically decreases. This is also the expected behavior, since the suspicion scores assigned to vertices do not change as we increase the threshold. There is an important point in the graph. At about $NT = 0.041$ FPR curve goes to 0 which means all the vertices (if any) in the solution set are attackers. At this point the $|SS|$ curve stays flat at the value of 1 until approximately $NT = 0.049$. This means that between $NT = 0.041$ and $NT = 0.049$ there was only one vertex in the solution set which is the attacker. This experiment showed that for this case our $SLANT$ algorithm we can find exactly who the real attacker was.

We repeated the same experiment for a network size of 1000. The graph at the bottom of figure 8.4.2 shows the results. We again see that both of the curves decrease monotonically as we move right on the x-axis. A similar behavior is repeated here by the two curves. Again at approximately $NT = 0.024$ the FPR curve goes to 0 while the $|SS|$ curve stays flat at the value of 1 until $NT = 0.025$. This again shows that at $NT = 0.024$ we can exactly identify the attacker.

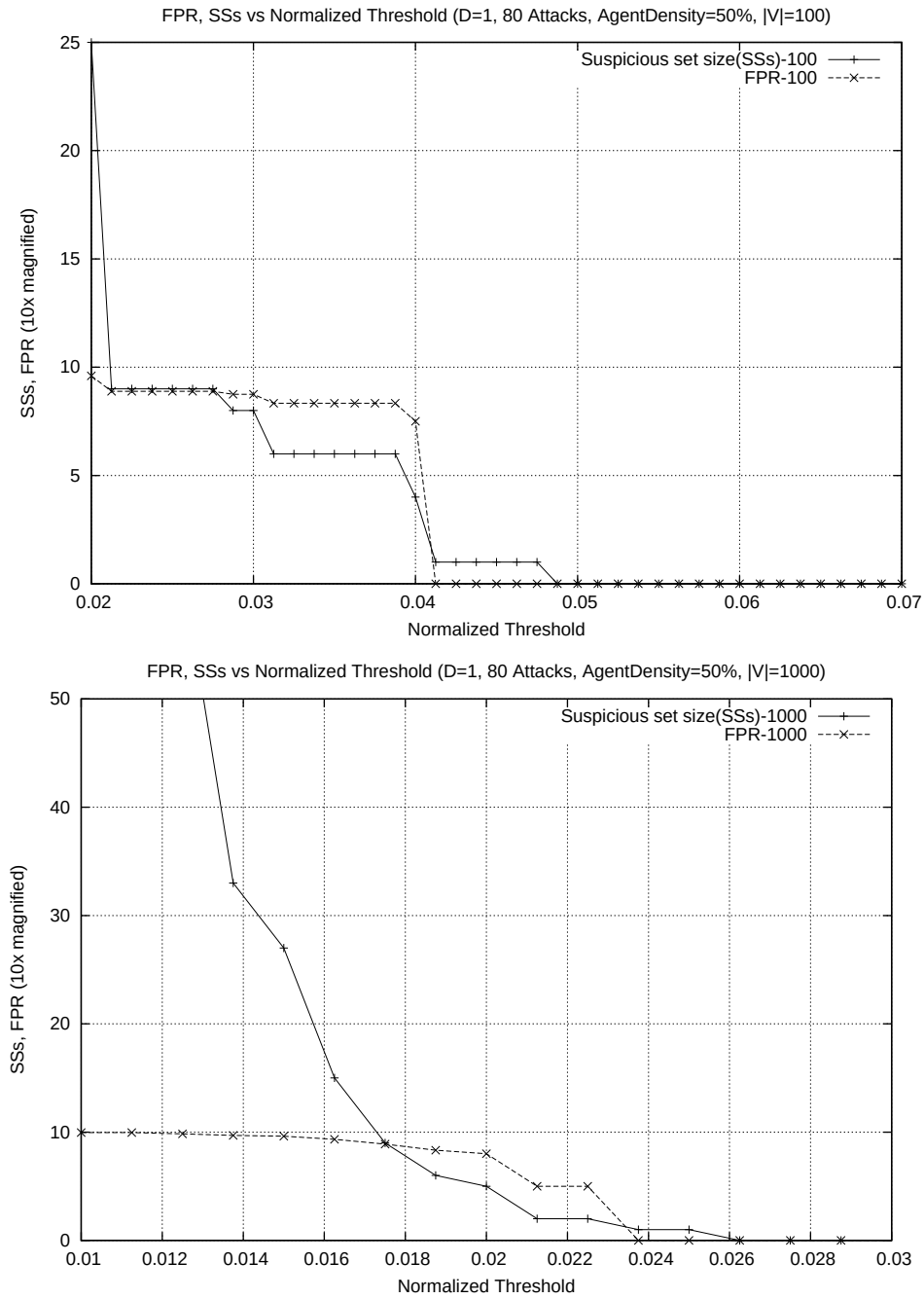


Figure 8.4.2: Experiments on grid network for varying network sizes

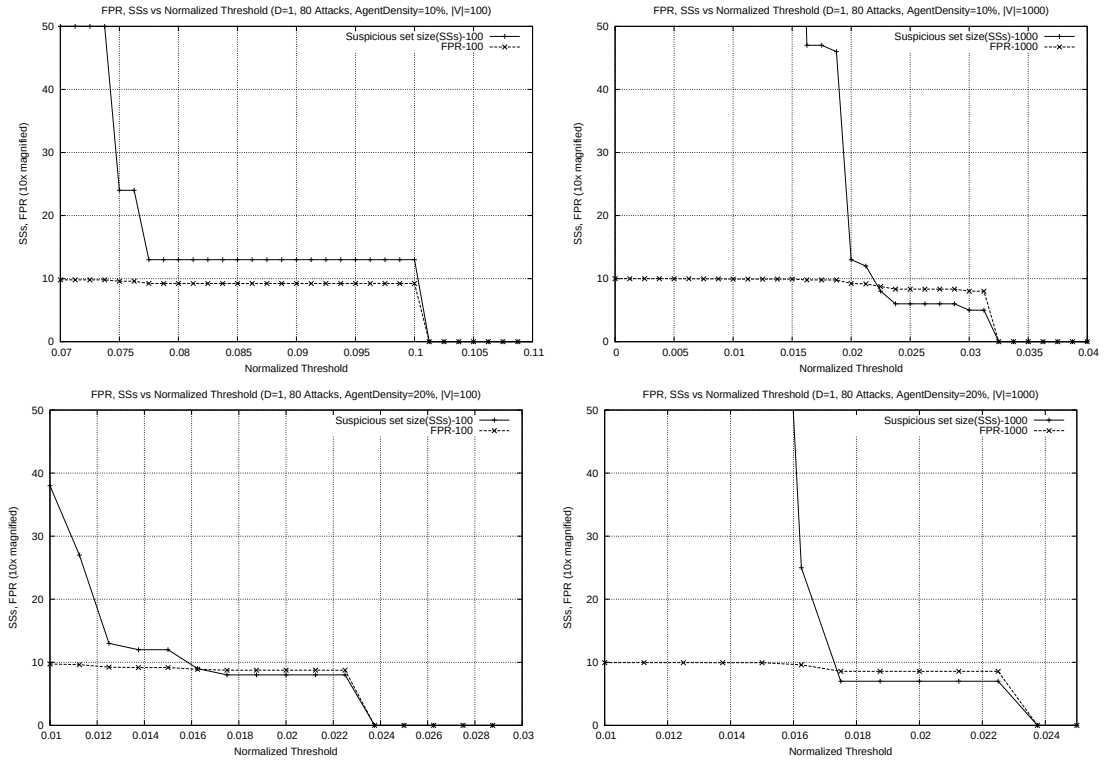


Figure 8.4.3: Experiments on grid network for varying agent density

8.4.3 Varying agent density

Purpose. The purpose of this experiment is to see how does the $SLANT$ algorithm behaves with different agent densities. We used grid networks, varying agent density levels between 10%,20%, and 50%, and network size $|V| = 100, 1000$. We fixed the number of attackers $|D|$ as 1, and the number of attacks n to be 80.

Results. Figures 8.4.3 show the experiment results for different agent densities and network sizes. The figures on the left show the results of experiments with network size of 100 and figures on the right show the results of the experiments with network size 1000.

The figures on the top row show the results when agent density is set to 10%, while the figures at the bottom row show the results when agent density is set to 20%. All the curves

in all of the figures show a monotonic decrease. This again supports the conclusion that the $SLANT$ algorithm behaves as desired under different agent densities. However, none of the figures show the same behavior as the previous results where FPR curve drops to zero but $|SS|$ curve stays at 1. This means that we can not exactly determine the attacker. This shows that we can not get closer to the attacker enough to exactly identify it. This is an expected behavior again. We can only get as close to the attacker as the closest agent surrounding it. So in a grid-like network we can not get closer to the attacker than an area determined by the attackers surrounding agents.

8.4.4 Varying the number of attacks

Purpose. The purpose of this experiment is to understand the effect of number of attacks n on $|SS|$ and FPR. We used grid networks, varying the network size $|V| = 100, 1000$ and varying the number of attacks $n = 10, 20, 40, 80$. We fixed the number of attackers $|D|$ as 1, agent density as 50%.

Results. Figure 8.4.4 shows the experiment results for different number of attacks and network sizes. The figures on the left show the results of experiments with network size of 100, while figures on the right show the results of the experiments with network size 1000.

The figures on the top row shows the results after 10 attacks, the figures in the middle row show the results after 20 attacks, the figures at the bottom row show the results after 40 attacks. All the curves in all of the figures show a monotonic decrease which again support the conclusion that the $SLANT$ algorithm behaves as expected under different number of attacks. Five of the figures show that there is point when the FPR is 0 and $|SS|$ is 1. Thus, in those cases we can exactly identify the attacker. The figure on top right shows that

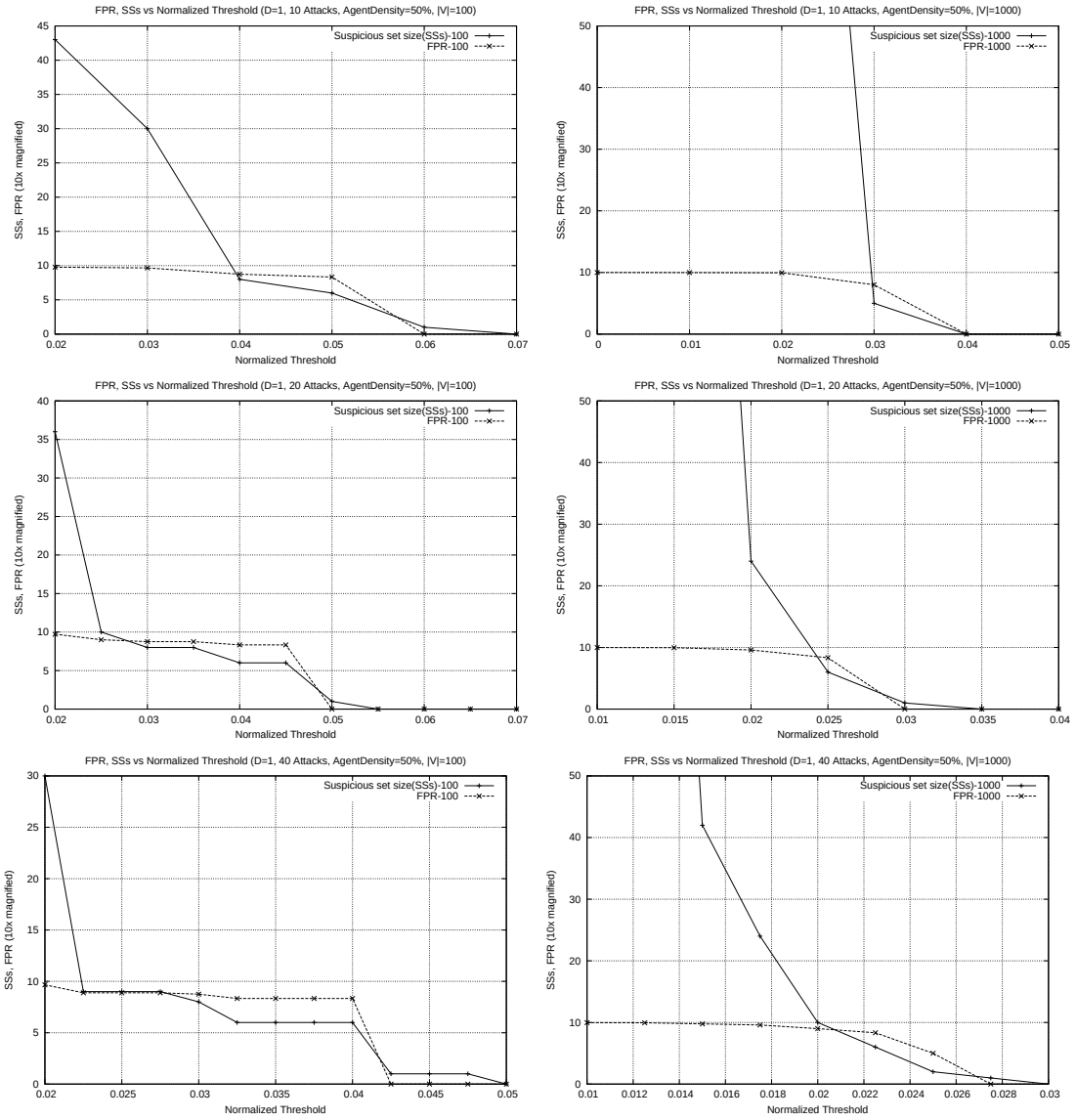


Figure 8.4.4: Experiments on grid network for varying number of attacks

when network size is 1000 and the number of attacks is 10 we can not exactly determine the attacker. However as the number of attacks increases we could exactly determine the attacker even in 1000 node network. These experiments demonstrate that as the network size increases we need to have more attacks to exactly determine the attacker. As we move down the rows we see that the NT value where FPR becomes zero moves towards 0. Consequently, as a result of this experiment, we can say that the $SLANT$ algorithm's success depends on the number of attacks n, and the minimum number of attacks required to exactly determine the attacker increases as the network size increases.

8.4.5 Localization in a Waxman network with one attacker

Purpose. The purpose of this experiment is to determine how well SLANT performs in Waxman type networks when there is one attacker. We use a Waxman network, varying $|V|$ between 100,200,400,800. We fix agent density at 15% and use greedy agent placement. We also fix the number of attackers $|D| = 1$, and the number of attacks $n = 80$.

Results. The figure 8.4.5 shows the performance of the algorithm in different sized Waxman networks when there is one attacker. There are four graphs in the figure. The graph on top right shows the result for a network size of 100 nodes. The graph on top left shows the result for a network size of 200 nodes. The graph on top right shows the result for a network size of 400 nodes. The graph on top right shows the result for a network size of 800 nodes.

All the of the curves decreases monotonically which agrees with earlier conclusions drawn about the far more restricted class of grid networks. In three of the graphs we can see that there is a point where $FPR = 0$ and $|SS| = 1$. Recall that this experiment is

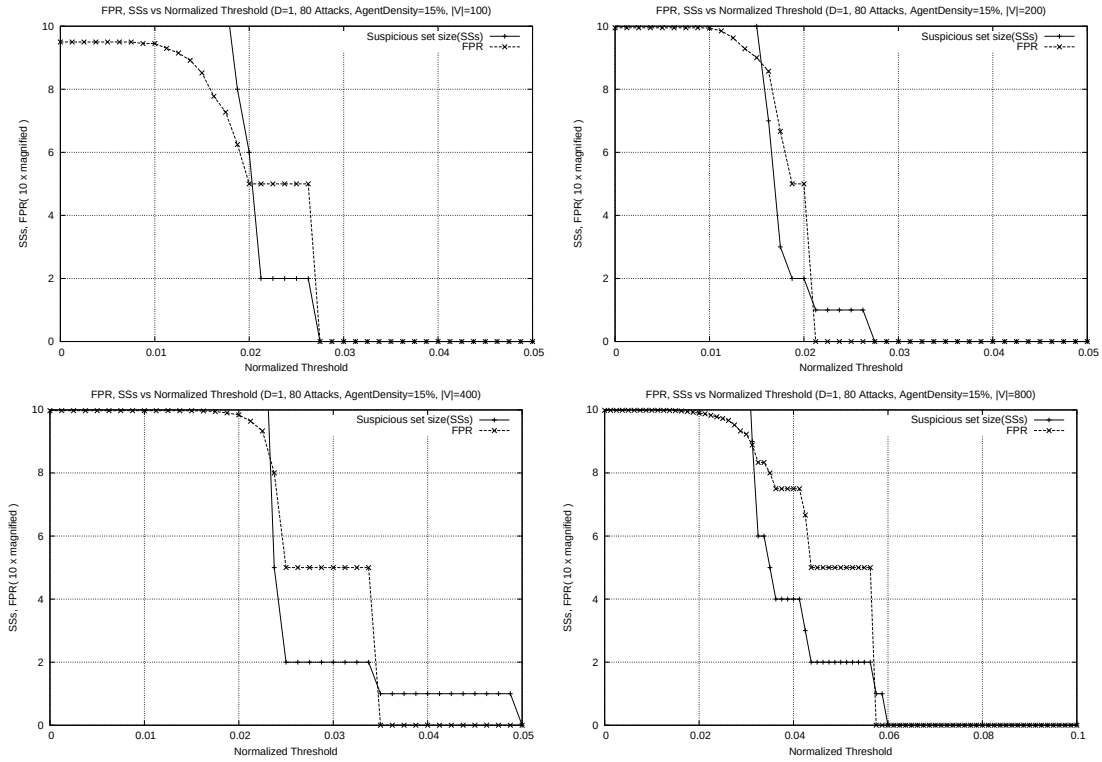


Figure 8.4.5: Experiments on Waxman networks with one attacker for varying network sizes

done for agent density of 15%. In grid-like networks, at this agent density, we were not able to determine the attacker exactly. Considering that the agent density of 50% is not realistic, with current results we believe that the $SLANT$ can be used in real networks to exactly determine attacker locations. More experiments are needed to ascertain this. In conclusion, $SLANT$ behaves favorably in Waxman network, and furthermore, it performs better in Waxman networks than in grid networks of comparable size (and using lower levels of agent deployment).

8.4.6 Localization in a Waxman network with many attackers

Purpose. The purpose of this experiment is to determine how well SLANT performs in Waxman type networks when there is more than one attacker. We use a Waxman network, varying $|V|$ between 100,200,400,800. We fix agent density at 15% and use greedy agent placement. We also fix the number of attackers $|D| = 10$, and the number of attacks $n = 80$.

Results. Figure 8.4.6 shows the performance of the algorithm in different sized Waxman networks when there are ten attackers. There are four graphs in the figure. The graph on top right shows the result for a network size of 100 nodes. The graph on top left shows the result for a network size of 200 nodes. The graph on top right shows the result for a network size of 400 nodes. The graph on top right shows the result for a network size of 800 nodes. All of the curves in all of the graphs but one decreases monotonically. Note that the FPR curve in the top right figure does not decrease monotonically. Unlike the cases when there is only one attacker, the FPR curve does not have to decrease monotonically when more than one attacker is present–there can be cases when it increases. This can happen in cases where a threshold increase removes only one vertex which is an attacker from suspicious

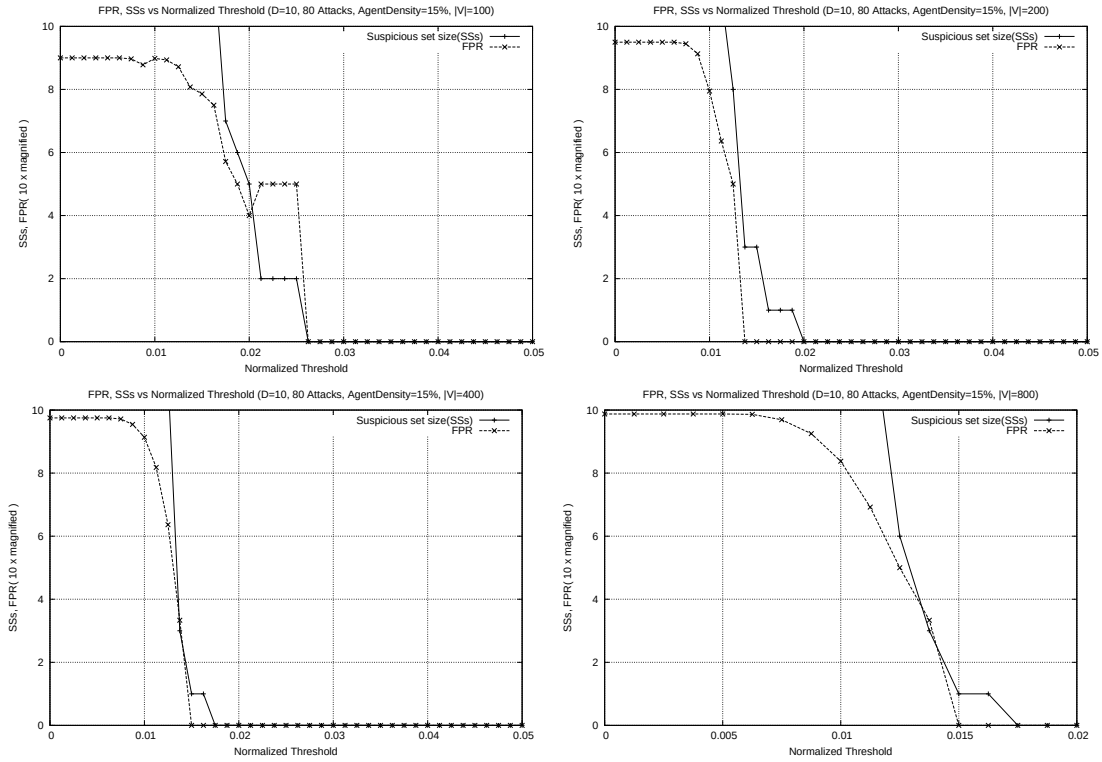


Figure 8.4.6: Experiments on Waxman networks with ten attackers for varying network sizes

vertices set. In this case since the percentage of the innocent vertices increases with respect to its percentage when the removed attacker was in the set, the FPR increases.

Other than that, in three of the figures, the FPR curve goes to 0 before the $|SS|$ curve reaches 0. Note that, unlike the one attacker case, the $|SS|$ curve can be greater than 1. this means we have exactly identified many of the attackers. Since this experiment was done for agent density of 15%, it appears that $SLANT$ can be used in real networks to exactly determine locations for a subset of the attackers. Although $SLANT$ can not exactly determine all of the attackers at once, with iterative implementation of the algorithm we can identify ever greater numbers of the attackers. This iterative application of SLANT is to be the subject of future research.

CHAPTER 9

CONCLUSION & FUTURE WORK

We have described a novel agent-based architecture for flow reconstruction, and described its application for obtaining a description of the structure and dynamics of distributed denial of service attacks. The proposed system is decentralized, and does not require a directory of agents. Unlike prior approaches, the proposed system provides a description of the structure and dynamics of traffic flows, tracing packets back from the victim towards the attackers. By providing structural information, the proposed system facilitates mitigation strategies close to the actual sources of attack traffic.

Through simulations, we showed that a significant fraction of attackers can be discovered even with very modest deployments of agents. We are able to get (on average) within 2-3 hops from the attackers. Through discrete event packet-level simulations, we verified that the state convergence is manifested over the entire range of parameter values.

We described two effective schemes for selecting the precise locations at which agents should be placed: M1-Greedy and M3-Greedy. We quantified the performance of these schemes and assessed their scalability. Through the experiments we saw that the greedy algorithms always perform significantly better than random placement, and provide good flow reconstruction capabilities even at modest agent deployment densities. We also showed that the two optimization criteria (M1 and M3) are concomitant: optimizing one tends to optimize the other. We also saw that these conclusions were consistent across different Waxman

networks of the same size and also the effectiveness of the schemes actually *improves* as network size increases. Taken together, these results point to the viability of the proposed system as a solution to flow reconstruction in general, and DDoS traceback in particular.

Finally, we developed a preliminary attacker localization scheme called SLANT, which shows promise for determining the exact locations of the attackers even at moderate deployment of agents. The success of SLANT indicates that it is possible to combine information from multiple attacks to localize the attacker positions. Although SLANT is able to provide a suspicion map encoding the most likely positions for attackers, the question of selecting the ideal suspicion threshold remains a topic of open research. We have reported promising initial findings relating the choice of suspicion threshold on attacker localization performance metrics.

9.1 Future Work

Throughout this research, we have made several simplifying assumptions. We intend to lift these assumptions, one at a time, in subsequent simulation studies:

- Routing tables are symmetric.
- Routing tables are consistent.
- Routing tables do not vary over time.
- Network structure does not vary over time.

The proposed system also assumes that distributed reflectors are not used by the attackers. We believe there is a way to modify the existing system to find the locations of the attackers

in the presence of reflectors. Roughly, in this case, the LDS must use a threshold on the variance in the number of distinct destination addresses targeted by a single source (within a finite time interval). If this variance exceeds a threshold, it is taken as indicative of an attacker who is launching an attack using reflectors; the spoofed source IP address reveals the intended victim which the reflectors are to target.

The experiments we used a traffic volume based LDS in which static thresholds are used to trigger alert generation. However, we know that there are better dynamic schemes than constant thresholding. For example [70] shows how one can sets the thresholds of underlying defense systems dynamically such that the total harm resulting from the attacks is minimized. It would be interesting to incorporate this type of LDS into the simulation experiments, permitting non-uniform threshold selection, and see the effects on system performance. Note that our experiments right now are the best-case results, since we assume perfect LDS operation.

There are also questions that we would like to consider in the context of applying the proposed agent placement algorithms on the present Internet topology:

- How should one place agents in the present Internet topology if one seeks to in maximize M1/M3 performance of the agent system?
- How should one place agents in the present Internet topology if one seeks to in maximize attacker localization performance of the agent system?

Here we note that the present algorithms for exact computation of E[M1] and E[M3] have a time complexity that prohibits resolution of the above two questions. Towards this end, we would like to:

- Consider GPU based implementations which can perform the exact computations in a reasonable time frame.
- Develop stochastic techniques to compute approximations for E[M1] and E[M3], thereby extending the range of network sizes for which agent placement is feasible.

The greedy algorithms for agent placement suggest several research questions:

- Why does randomized IGAP-k placement not get significantly better as k is increased? Why does randomized IGAP-k placement not get significantly better as the number of iterations is increased? Clearly, the solution attains a value that is locally optimal, but how far is this from global optimum?
- While the general computation of optimal agent placement appears exponential in network size, are there polynomial time algorithms for optimal agent placement for more restricted families of graphs?
- Is there an agent placement algorithm that outperforms randomized IGAP-k?

The approach of attacker localization presents many questions and opportunities for further research. Specifically,

- How should one select the suspicion threshold τ in order to obtain a low false positive rates while still maintaining a significant suspicious set size?
- Are there better choices for the multi-agent information combining function σ which consider metric geometry within the cone?
- Are there better choices for the multi-attack information combining function which weight more recent attacks over older attacks? To what extent can this be used to

lift our current research assumption that the membership of the set of attackers does not change over the sequence of attacks during the attacker localization process?

- Can one bridge the agent placement problem with the notion of attacker localization. In other words, how can one formalize the problem of placing agents in a manner which optimizes the attacker localization potential of the agent system?

We have largely ignored the possibility of attacks against the agents themselves, in part by separating the notion of recording stations (whose identities are public) from agents (whose identities are secret). Bringing this assumption into scrutiny raises the possibility of further extensions to this research:

- Given a fixed set of agents, how can one design a placement of attacker nodes, such that the expected M1/M3 performance of the system is minimized? This dual "Attacker placement problem" provides a lower bound on system performance in the setting where agent locations have been compromised. To what extent can this be used to lift our current research assumption that attacker locations are random and independent of agent locations (i.e. agent locations are held secret).

REFERENCES

[1] About clean slate, 2010. [Online; accessed 05-January-2010, Available online: http://cleanslate.stanford.edu/about_cleanslate.php].

[2] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. *ACM Trans. Interet Technol.*, 5(2):299–327, 2005.

[3] C. Advisory. Smurf IP denial-of-service attacks, 1998.

[4] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities. *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[5] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities. *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[6] D. K. Angelos, M. Vishal, and R. Dan. SOS: secure overlay services. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 61–72, New York, NY, USA, 2002. ACM.

[7] K. Argyraki and D. Cheriton. Network capabilities: The good, the bad and the ugly. In *Fourth Workshop on Hot Topics in Networks*, November 2005.

[8] G. B., M. C., , and G. B. ISP security-real world techniques. presentation, nanog. Technical report, 2001. [Online; accessed 05-January-2010, Available online: http://www.nanog.org/meetings/nanog36/presentations/greene.ppt].

[9] Barford, Paul, Kline, Jeffery, Plonka, David, Ron, and Amos. A signal analysis of network traffic anomalies. In *IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, pages 71–82, New York, NY, USA, 2002. ACM.

[10] Bellovin. Cert advisory ca-1996-26. Cert Advisory, 1996.

[11] Bellovin. ICMP traceback messages. RFC draft, September 2000.

[12] D. Bernstein. Syn cookies, 1996. [Online; accessed 05-January-2010, Available online: http://cr.yp.to/syncookies.html].

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[14] J. D. Brutlag. Aberrant behavior detection in time series for network monitoring. In *LISA '00: Proceedings of the 14th USENIX conference on System administration*, pages 139–146, Berkeley, CA, USA, 2000. USENIX Association.

[15] Burch and Hal. Tracing anonymous packets to their approximate source. In *LISA '00: Proceedings of the 14th USENIX conference on System administration*, pages 319–328, Berkeley, CA, USA, 2000. USENIX Association.

[16] comScore Inc. Global internet audience surpasses 1 billion visitors, according to comscore. Technical report, comScore Inc., 2009. [Online; accessed 05-January-2010, Available online: http://www.cidr-report.org/as2.0/].

[17] D. Cotroneo, L. Peluso, S. Romano, and G. Ventre. An active security protocol against dos attacks. *Computers and Communications, IEEE Symposium on*, 0:496, 2002.

[18] D. Danchev. Iranian opposition launches organized cyber attack against pro-ahmadinejad sites. *zdnet.com*. [Online; accessed 05-January-2010, Available online: http://blogs.zdnet.com/security/?p=3613].

[19] O. Demir. A survey of network denial of service attacks and countermeasures. Technical report, City University of New York, Computer Science Department, 2009.

[20] D. Dittrich. The stacheldraht distributed denial of service attack tool, 1999. [Online; accessed 05-January-2010, Available online:http://staff.washington.edu/dittrich/misc/stacheldraht.analysis].

[21] W. Eddy. TCP SYN flooding attacks and common mitigations. RFC 4987 (Informational), August 2007.

[22] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred. Statistical approaches to DDoS attack detection and response. *DARPA Information Survivability Conference and Exposition,*, 1:303, 2003.

[23] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing, 2000.

[24] Gil, T. M., Poletto, and Massimiliano. MULTOPS: a data-structure for bandwidth attack detection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 3–3, Berkeley, CA, USA, 2001. USENIX Association.

[25] V. D. Gligor. A note on denial-of-service in operating systems. *IEEE Trans. Softw. Eng.*, 10(3):320–324, 1984.

[26] D. Goodin. Teen hacker confesses three-year crime spree, 2008. [Online; accessed 05-January-2010, Available online: http://www.theregister.co.uk /2008 /11/19/dshocker_pleads_guilty/].

[27] W. Haining, Z. Danlu, and K. G. Shin. Detecting SYN flooding attacks. In *In Proceedings of the IEEE Infocom*, pages 1530–1539, 2002.

[28] J. Hawkinson. Guidelines for creation, selection, and registration of an autonomous system (as). RFC 1930 (Best Current Practice), March 1996. [Online; accessed 05-January-2010, Available online: http://tools.ietf.org/html/rfc1930].

[29] S. A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473, 2000. PMID: 11130924.

[30] T. R. Irirangi and O. Aotearoa. Marshalls internet still affected after cyber attack, 2008. [Online; accessed 05-January-2010, Available online: http://www.rnzi.com/pages/news.php?op=read—&id=40547].

[31] A. L. N. Kim, S. S. Reddy. Image-based anomaly detection technique: Algorithm, implementation and effectiveness. *IEEE journal on SELECTED AREAS in COMMUNICATIONS*, 24(10):1942–1954, 2006.

[32] J. Kirk. Georgia president's web site falls under ddos attack, 2008. [Online; accessed 05-January-2010, Available online:http://www.networkworld.com/news/2008/072108-georgia-presidents-web-site-falls.html].

[33] J. Kirk and I. N. Service. Two europeans charged in us over ddos attacks, 2008. [Online; accessed 05-January-2010, Available online: http://www.pcworld.com/businesscenter/article/151829/two_europeans_charged_in_us_over_ddos_attacks.html].

[34] M. LEE. Reply-all e-mail storm hits state, 2009. [Online; accessed 05-January-2010, Available online: http://www.huffingtonpost.com/2009/01/10/replyall extscore email extscore storm extscore hits_n_156856.html].

[35] J. Leyden. OcUK puts 10k bounty on the heads of ddos varmints, 2009. [Online; accessed 05-January-2010, Available online: http://www.theregister.co.uk/2009/01/22/ocuk_ddos_reward/].

[36] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang. SAVE: Source address validity enforcement protocol. In *In Proceedings of IEEE INFOCOM 2002*, pages 1557–1566, 2002.

[37] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *SIGCOMM Comput. Commun. Rev.*, 32(3):62–73, 2002.

[38] A. B. Mail and A. Back. Hashcash - amortizable publicly auditable cost-functions. Technical report, 2000.

[39] T. R. Malthus. *An Essay on the Principle of Population 2 volume set*. Cambridge University Press, 1990.

[40] G. H. Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, pages 1045–1079, 1955.

[41] S. Microsystems. Class hashmap, 2003. [Online; accessed 12-January-2010, Available online:http:// www.networkworld.com/news/2008/072108-georgia-presidents-web-site-falls.html].

[42] S. Microsystems. Class hashset, 2003. [Online; accessed 12-January-2010, Available online:http://java.sun.com/j2se/1.4.2/docs/api/java/util/HashSet.html].

[43] J. Mirkovic. *D-ward: source-end defense against distributed denial-of-service attacks*. PhD thesis, 2003. Chair-Gerla,, Mario and Chair-Reiher,, Peter.

[44] G. Mohsen, R. Ammar, K. Bilal, and A.-F. Ala. *Network Modeling and Simulation: A Practical Perspective*. Wiley-Interscience, 2010.

[45] A. Networks. Arbor networks worldwide infrastructure security report. Technical report, Arbor Networks Inc., 2008. [Online; accessed 05-January-2010, Available online: http://www.arbornetworks.com/en/docman/worldwide-infrastructure-security-report- volume-iv-2008-/download.html].

[46] K. Park and H. Lee. On the effectiveness of probabilistic packet marking for IP traceback under denial of service attack. In *INFOCOM 2001*, pages 338–347, 2001.

[47] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, New York, NY, USA, 2001. ACM.

[48] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *SIGCOMM Comput. Commun. Rev.*, 31(3):38–47, 2001.

[49] T. Peng, C. Leckie, and K. Ramamohanarao. Protection from distributed denial of service attack using history-based IP filtering. In *Communications, 2003. ICC '03. IEEE International Conference*, pages 482–486, 2003.

[50] T. Peng, C. Leckie, and K. Ramamohanarao. Survey of network-based defense mechanisms countering the DoS and DDoS problems. *ACM Comput. Surv.*, 39(1):3, 2007.

[51] R. Powers. *Prisoner's Dilemma*. Beech Tree Books, 1988.

[52] V. R and E. G. DNS amplification attacks. Technical report, 2006. [Online; accessed 05-January-2010, Available online: http://www.isotf.org/news/DNS-Amplification-Attacks.pdf].

[53] B. R.B., Kim.H, R. b., and T. A. A novel approach to detection of denial of service attacks via adaptive sequential change point detection methods. In *Proceedings of 2001 ieee systems Man and Cybernetics Information Assurance*, 2001.

[54] T. C. Report. CIDR report for 5 jan 10. Technical report, www.cidr-report.org, 2010.

[55] C. Reporter. Network solutions under large scale ddos attack, millions of websites potentially unreachable, 2009. [Online; accessed 05-January-2010, Available online: http://www.circleid.com/posts/20090123_network_solutions_down_ddos_attack].

[56] C. L. Schuba, I. V. Krsul, M. G. Kuhn, E. H. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *In Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223. IEEE Computer Society Press, 1997.

[57] Snoeren and A. C. Hash-based IP traceback. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14, New York, NY, USA, 2001. ACM.

[58] S. Stefan, W. David, K. Anna, and A. Tom. Practical network support for IP traceback. *SIGCOMM Comput. Commun. Rev.*, 30(4):295–306, 2000.

[59] Stone and Robert. Centertrack: an ip overlay network for tracking DoS floods. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 15–15, Berkeley, CA, USA, 2000. USENIX Association.

[60] Z. Sun, D. He, L. Liang, and H. Cruickshank. Internet qos and traffic modelling. *IEE Proceedings - Software*, 151(5):248–255, 2004.

[61] G. Urdaneta, G. Pierre, and M. van Steen. A survey of DHT security techniques. *ACM Computing Surveys*, 2009. Available online:http://www.globule.org/publi/SDST_acmcs2009.html.

[62] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[63] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 303–314, New York, NY, USA, 2006. ACM.

[64] B. M. Waxman. Routing of multipoint connections. pages 347–352, 1991.

[65] P. D. Williams, K. P. Anchor, J. L. Bebo, G. H. Gunsch, and G. D. Lamont. CDIS: Towards a computer immune system for detecting network intrusions. In *RAID '00:*

Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection, pages 117–133, London, UK, 2001. Springer-Verlag.

[66] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless internet flow filter to mitigate ddos flooding attacks. *Security and Privacy, IEEE Symposium on*, 0:130, 2004.

[67] D. K. Y. Yau, J. C. S. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Trans. Netw.*, 13(1):29–42, 2005.

[68] H. Young, H. Bradley, A. Dan, A. Emile, L. Matthew, and S. Colleen. The IPv4 routed /24 as links dataset11/15/2009.

[69] H. Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.

[70] C. C. Zou, N. Duffield, D. Towsley, and W. Gong. Adaptive defense against various network attacks. In *In Proceedings of the Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI*, pages 69–75. IEEE Press, 2005.

INDEX