Aminon: The DNS Capabilities Gatekeeper

Charlie Lovering, Derek McMaster, Akshit (Axe) Soota, [Redacted]

***Abstract*—Unwanted traffic is an issue currently dealt with from the end-host outward. Currently, numerous solutions and ideas have been investigated from points in the network considered more unique to each endpoint and not necessarily from a more centralized scheme. Although many of these are effective, we set out to propose a system that protects the user instead of relying on the user to protect itself. Our implementation, named *Aminon*, attempts to mitigate users of this issue by leveraging the Domain Name Service (henceforth DNS) query as a capability token to be granted access to securely communicate with various web servers. This approach, based off of the academic paper *On Building Inexpensive Network Capabilities*, introduces a system taking advantage of the currently in-place DNS infrastructure, but uses a user's behavior as their means of protection. This capability system will require more steps in opening and maintaining lines of communication through DNS, but will be shown to greatly reduce the effectiveness of currently deployed, web-based attacks such as IP Scanning. In this paper we not only submit a unique system to deal with unwanted traffic, but also evaluate the costs associated with it and the performance differences compared to the current DNS system.**

I. Introduction

UNWANTED traffic is an issue that affects every person who uses the internet. Whether it takes the form of a malicious attack, spam email, or recurring advertisements, unwanted traffic obfuscates an end user's experience and impacts the ever-growing community of people coming online. Many researchers have proposed innovative and creative techniques and systems to solve this persevering issue. In terms of proposals for DNS systems, there is an aggregate tendency to examine the system outwards-in, from the point-of-view of an end user. Many things such as cost and overhead promote this idea because replacing the current DNS is intractable. Our approach is unaffected by these concerns because we leverage the current DNS infrastructure against itself in attempts to solve this problem. Over the course of the internet's lifespan and the increasing complexities and applications we employ to control it, we have examined behavioral patterns for both attackers (the adversary or malicious user) and non-malicious users (the client). One recurring tendency users exhibit is accessing webpages through named DNS queries, or queries where the users' queries using the alphanumeric representation of the server that they're attempting to access. Interestingly, seasoned attackers typically exhibit the exact opposite behavior, querying servers using their Internet Protocol addresses (IP). Based on this behavior we build *Aminon*, a DNS gatekeeping system which takes advantage of these behaviors to provide security to the end user and the destination web server. The key benefits of this implementation are the circumventing of any infrastructure costs possibly incurred by changing DNS systems and the fact that it requires no education of the population to become effective, it simply adapts to current user trends. While this system is not completely invulnerable to any type of malicious attack, it will deter some of the more prevalent attacks such as IP scanning and **D**enial **o**f **S**ervice (DoS) attacks. Most notably, in mitigating these attacks it provides the outcome of minimizing unwanted traffic benefitting the internet community as a whole. We start out by introducing *Aminon* as a system. We set out the assumptions of the system we implemented on, along with the threat model we keep in mind during integration. After the scene is set, we lay out the system model to give the reader a visual and a precedent to build off of throughout the evaluation and results of implementation. The system architecture comes after the model and will explain how we provisioned the four virtual machines (henceforth VMs) and what tools and techniques we employ to accomplish this design. When the foundation is laid, we take a deep dive into all of the components of the system working down into fine-grained details of interactions and implementation specifics. After the reader understands how the system interacts and communicates, we aim to explain how the system is evaluated and compared to the current DNS implementation statistics. This includes the key concerns we had when implementing and the methodology we followed to circumvent them. When researching the key concerns, we aggregated some important metrics we aimed to compete with using *Aminon* versus an ordinary DNS infrastructure. These metrics span all components of the system and include vital timing comparisons such as DNS query duration. After explaining the reasoning behind our evaluation approach we submit to the reader the results of our system's implementation. We provide graphs, tables, and explanations as to the comparison of the results to the projected metrics. When the reader has a full view of the scope of the project, including the setup, evaluation, and results we conclude with a charge of the viability of this approach.

II. Assumptions

We make the following persisting assumptions about the system we will be implementing on:

- *Peer-to-peer connections:* Such connections will not be tunneled through our system and shall be allowed to pass without need to test for capabilities.
- *Browser Pinning:* We assume browser pinning will be negligible. We will use command line HTTP libraries rather than browsers with "no cache" options to be sure of mitigating this concern.
- *All manners of connections:* It may seem like a trivial thing to point out, but we will assume that there are authentic and malicious users on the network attempting to access any of the other components on the system.
- *Authoritative DNS Resolver:* Our system will be running on a closed network, and our DNS resolver will act as

an authoritative DNS system instead of possibly passing through many DNS servers to resolve. This is done for simplification and for ease in checking our proof-of-concept.

III. Threat Model

We will examine each of the following threats we believe we will encounter during the system's lifespan:

- *IP Scanning*
  A lot of modern-day attacks involve brute-force scanning though all possible IP addresses available in the public address space. By implementing DNS Capabilities, we thwart such attacks by forcing users to first run a DNS lookup which grants them a capability to communicate.
- ***D**enial **o**f **S**ervice (DoS) Attacks*
  A denial of service attack will not be effective on a server masked by a DNS gatekeeper. All attacks will be by default routed locally towards a honeypot. If a query is not granted a capability it will fall through the rest of the functionality and into the honeypot otherwise it will be properly handled. A DoS is generally not generated by DNS queries, and thus the attacker will not be granted a capability because it did not do a look-up and we will implement other requirements in the case they use DNS lookups or if they try to request many times.
- *IP Spoofing*
  IP spoofing will not be effective because the capability decision is granted less on the IP but more on the characteristics of the request. If caching or storing previous good connections is exploited to try to effectively get another capability subsequent countermeasures for DoS attacks such as rate limits are employed through *iptables* rules to handle the packets.
- ***D**enial **o**f **C**apability (DoC) Attacks*
  The logical next attack would be a denial of capability attack where an attacker would flood a DNS server with requests for a capability, and thus prevent legitimate users from doing the same. Theoretically this could pose a problem, but it is unlikely to work in practice. The decision to grant a capability or not is incredibly lightweight, and thus a DNS server can serve many requests quickly making it far more difficult to overwhelm than most application servers.

IV. System Model

Our system will consist of 4 main actors. This consists of users (both malicious and non-malicious), a target server (the asset server), a router with a configured NAT implementation, and a DNS server with the gatekeeper privileges to grant capabilities.

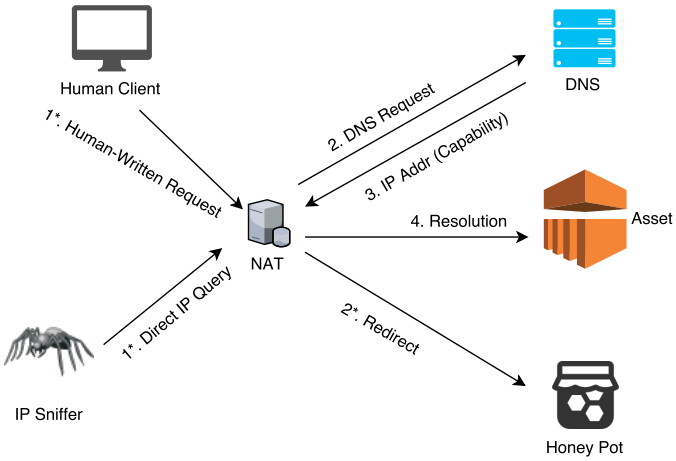

Fig. 1: Aminon's System Model

V. Implementation Strategy

This section will provide an overview of the systems, tools, and techniques used to implement the *Aminon* system. For specifics, review Section VI as each components' model and critical functions are defined.

A. System Architecture

We have limited resources, (4 virtual machines) to test this system. Each virtual machine played one of the following roles:

- *DNS Resolver and NAT:* This will help the client to map host names to the fluxing public IP of the server to be protected. The resolver will map all unwanted traffic to this honeypot. The NAT will reside in the same virtual machine. All the traffic from the client is routed through the NAT translation tables and will map the client to the destination server on the network.
- *Asset Server (to be protected):* This is the server that is being protected from unwanted/bot-based traffic.
- *Client and Attacker:* This will be the user that is trying to connect to the server that is being protected. We will use this machine to also attack the network, and evaluate its effectiveness against automated attacks.
- *Honeypot Server:* The last VM will be running a web-server functioning as a honeypot server.

We have root access to all 4 of the virtual machines. Executions on the virtual machines is done over SSH and a user was authenticated using a public-private key fingerprint that was setup during the setup of the virtual machines.

B. Tools and Techniques

- *bind9 / named:* An authoritative DNS name server provided as a part of the BIND9 framework, this will be used to resolve the client's requests while vetting them against the NAT.
- *rndc:* This name server control utility is used for the operations of the named server such as checking its status

as well as starting, stopping, and restarting the server while editing the configuration files.
- *iptables:* An administration tool for IPv4/IPv6 packet filtering and NAT translation tables. This will be used to randomize the mapping of IP aliases to the actual IP address in an attempt to require the client to type the actual domain name so as eliminate the success of IP brute force sniffing.
- *nmap:* Analyzes raw IP packets in unique ways in order to determine what hosts are available on the network and what services they provide. This will be used to sniff for IP addresses and open ports.
- *dig (domain information groper):* A flexible tool for querying the DNS name servers.
- *wireshark:* A tool to interactively dump and analyze network traffic. This will be used to test packet flow on the network.
- *tcpdump:* Another method of dumping traffic on a network from the command line, also used for packet testing purposes.
- *route:* A Linux program used to show or manipulate the IP routing table. This can be used to prove network connectivity as coordinated by *iptables*.
- *traceroute:* This tool is designed to print the route packets take to network host. Used for testing purposes.
- *netfilter_queue:* Used to delay DNS until NAT is configured properly. It will handle and manipulate packets transmitted between DNS and NAT.
- *wget:* A utility for "non-interactive download of files from the Web. It supports HTTP, HTTPS, and FTP protocols" [1]. This will be used by the client to validate accessing the domain name.
- *lynx:* A general purpose distributed information browser for the World Wide Web. For all intents and purposes we will use this to test the puzzle verification for exit from the honeypot server (captcha), in the case that a legitimate user ends up in the honeypot (in which case it is necessary to submit the answer to the server).

VI. Components

For each component we outline its purpose, how it interacts with the other components, and our strategy for implementation and highlights of important functions. Our system consists of the following components:

A. DNS Server

1) Purpose: The DNS resolver is the core component of *Aminon's* functionality. It performs the physical indexing of the queries and determines a fluctuated public IP to map the client to the asset server. Specifically, we implement how *Aminon* determines whether or not to return the capability. Fortunately, this is done separately from the rest of the functionality necessary, and thus we can experiment with different policies quickly. **Listing 1** briefly shows how we can quickly switch how we address new, incoming requests, by changing the configuration of the function generally named *policy_decision*. The DNS resolver will resolve domain name requests when the *policy_decision* function reasons that the request originates from a legitimate user by returning that client the current public address of the server.

2) Interface and Interactions: The DNS Resolver runs on a separate Virtual Machine along with the NAT and will act as the root DNS server. All DNS requests made within the network should be directed to this resolver. It is then in our control if the user is given access to the requested resource. There are two scenarios:

1) If the client is granted access to the resource being requested, the DNS returns a mapping to a server's random public IP. In doing so, the NAT is notified of the fact that one of the randomly fluxed IPs were returned. The NAT then makes the necessary changes to its tables to allow the client to connect to the asset if it were requested with the public IP that was returned.
2) If the client is not granted access to the resource being requested, the DNS returns a mapping to the honeypot server.

3) Implementation Details:

a) Strategy: We use *named* from *BIND9* as the main tool to implement a DNS server. The DNS server is configured in a file *named.conf*, while the functionality such as analyzing, stopping, and restarting named is controlled via *rndc*. All outbound DNS responses are held in *netfilter_queue* while the NAT is notified about the public IP being given out. Specifically, DNS response packets are immediately aggregated in a queue in the kernel to process and send out to the client. In order to coordinate with the NAT and inform it that this public address is being sent to a client, we use *netfilter_queue* to remove this packet from the queue and wait for a confirmation from the NAT.

b) Functions:

Listing 1: Abstract DNS Resolver

```
def dns_request(req, dom):

   /* determine if we should give the
       capability */
   capability = policy_decision(req)
   /* log the request */
   log_request(req, capability)

   if capability:
      real_address = lookup_addr(domain_name)
      response = dns_reponse(real_address)

      /* inform the nat */
      netfilterqueue_wait(response,
          report_address_dns(real_address))

      return response
   else:
      honey_address = get_honeypot(req)
      return dns_reponse(honey_address)
```

B. NAT

1) Purpose: The NAT will act as the capability enforcer for *Aminon*. The table entries in the NAT hold mappings from

the public IPs handed out to the clients to the private IP of the asset server they correspond to.

2) Interface and Interactions: We ran the NAT on the same virtual machine as the DNS server. This is done because there is a close coupling between the two: the DNS will directly communicate back-and-forth with the NAT to act as a gatekeeper. The NAT will maintain two tables and update various mappings based on their time-to-live's (TTLs). The separate tables stored in the NAT are Table "N" (for new incoming connections) and Table "E" (for existing and established connections or their fluctuated IP addresses from the DNS server). By default, the NAT table N would contain a mapping of all fluctuated IP addresses generated until the date and point to the honeypot server. When the DNS notifies that one of the public IPs was handed out to a client, the NAT moves that public IP address from table N to table E and maps that IP to the actual private IP of the target server. In doing so, the NAT also assigns a TTL value to that entry. The system will remove the mappings once the TTL runs out but we will add a small buffer time before the mapping truly gets erased. It will be a very short buffer but will make sure the client can attempt a query in a reasonable amount of time without having the mapping erased and needing to take the time to resolve another query for a vetted, non-malicious client.

3) Implementation Details:

a) Strategy: We will be using the program *iptables* to control and configure our NAT infrastructure. We will also be using a few built-in, Python libraries to complete this implementation.

b) Functions:

Listing 2: Update NAT Function

```
def update_connections():
    buffer = 5;
    scan through the TTLs in the existing
        connections
    /* buffer is a limbo period for client to
        try mapping again and not have to
        requery DNS server */
    if (TTL + buffer) == 0:
      clear the entry
    else:
       move the public IP back up to the pool
           of available IPs
```

Listing 3: Function to create a mapping in the NAT

```
def insert_map_connection(pub_ip, priv_ip):
   mapping = update_iptables(pub_ip, priv_ip)
   /* Chosen TTL = 300 seconds */
   set_ttl(mapping, 300)
   insert(mapping)
   /* Add to active mappings */
   active_mappings.append(mapping)
   /* Delete expired mappings */
   update_connections()
```

c) Example Tables for the NAT:

Listing 4: Visualized Outline of NAT Table

```
Table N (ports excluded):
    Public IPs      Honeypot IP
    150.120.43.1    10.1.1.1
    150.120.43.2    10.1.1.1
    ...             ...
Table E (ports excluded):
    Public IPs      Mapped Private IP    TTL
    160.120.43.2    192.15.10.4          360
    150.130.42.5    192.15.10.1          225
    ...             ...                  ...
```

C. Honeypot

1) Purpose: The goal of setting up the honeypot is to redirect users running IP scans or lacking the capability to access a requested resource to a proof-of-concept web server at a random port. This could be extended to analyzing incoming web traffic and learning about the kinds of attacks that the network is coming under or logging repeat offenders for analysis purposes.

2) Interface and Interactions: The honeypot will be a simple Python server running on a random port on a virtual machine. The NAT table entries will ensure that all unassigned IP addresses will be internally mapped to the honeypot server running on the DNS Server. Every time a user visits the honeypot, their IP address, User-Agent and the time of the day will be logged to a file. This could help us analyze incoming web traffic to the honeypot in the future.

3) Implementation Details:

a) Strategy: We implement the honeypot server using a python script. We make sure the honeypot server listens on a very specifically numbered port as to be sure a non-malicious user won't be redirected there on anything but a failing of our system.

b) Functions:

Listing 5: Honeypot Service

```
def web_request(request):
  /* Get user info */
  info = [request.source_ip, \
          request.time, \
          request.headers["User-Agent"]]
   /* Add information to server logs */
   write_to_env(ENV["log"], info)
   /* Deliver honeypot page */
   server_page(ENV["honeypot_path"])
```

D. Escape from Honeypot

1) Purpose: Caching is a common issue which could cause the user to be redirected to a honeypot. Under such circumstances, we do not want to restrict access to resources from legitimate users. To allow the user uninterrupted access to the requested resource, the user would have to fill out a captcha to prove that they are not an automated attacker, and then the DNS gatekeeper will re-validate their session.

2) Interface and Interactions: The escape from honeypot will be a small feature of this system that allows a user to leave the honeypot and go to its original target of the DNS query.

3) Implementation Details:

a) Strategy: The honeypot will be a simple web server that logs information of incoming traffic. The captcha itself will be a simple text correlation question, that serves as a security feature but we concede there may be potential for a security leak there that the honeypot is attempting to accomplish. For example, there will be in plaintext the word 'nyne' and the question, 'what number am I thinking of?' (to which the user would be expected to write the word 'nine' although we may consider accepting '9'). The user will input an answer to a form, and upon submitting the form the honeypot will verify the answer, as shown below in **Listing 6**. We expect the captcha questions to be text based and easy to answer without much thought. We aim to formulate the questions to so a basic botnet cannot parse the questions but we will also allow a small number of retry attempts to facilitate confusion for users. 6. If its valid the server will contact the DNS, log the success, and reroute the user to its original goal.

b) Functions:

Listing 6: Server-Side Captcha Validation

```
def validate(user_response):
  /* Get the captcha and user's destination */
  captcha = get_captcha()
  user_dest = get_user_dest()
  /* Check for response validity */
  if correct(get_answer(captcha),
      user_response):
    /* Log the user successfully completed
       the captcha */
    log(user, True)
    /* Redirect the user to the actual asset
       server */
    redirect_user(user_dest)
  else:
    /* Log the user failed the captcha */
    log(user, False)
```

E. Attacker and Client

1) Purpose: This components attempt to access the asset server. The attacker simulates someone attempting to automatically scan the IP space to find vulnerable or exposed servers. The client will be a legitimate, non-malicious user trying to gain access to the asset server.

2) Interface and Interactions: The attacker and client are essentially the same component in terms of setup and tools and thus reside on the same virtual machine. Since we will essentially be removing a NAT translation from client to the DNS server, we will hard-code the location of the DNS server on the client and leave the resolution and granting of capabilities to the DNS server as defined above.

3) Implementation Details:

a) Strategy: The attacker will use *nmap* to identify an IP space and then they will continue to probe the identified IP space with a script that continually pings an IP space. A user will attempt to reach the server using a terminal based-browser such as *lynx*.

b) Functions:

Listing 7: User

```
lynx (server_domain_name)
if SUCCESS:
  explore page
 else:
  try captcha
```

VII. Policy and Communication Protocol

A. Capability Granting

The client will attempt to query the DNS server to translate the domain name into an IP address, at which point the NAT table will grant the client the capability to do so. However, if the client doesn't request the DNS server to translate, but instead attempts to simply connect directly to an IP address, the NAT table will instead route the traffic to the honeypot.

B. Fluxing Strategy

We plan to flux the public IP given by the DNS server every time the DNS server is queried for it. This is because when a user is granted a capability and connects to the server, it will be moved to an established connection and live off its TTL. Then the DNS server will randomly generate a fluxed IP for the server the user is attempting to query. As a way to mitigate severed connections, every time the user queries a name resolution while the TTL is still valid, it will be refreshed. If the TTL for the mapping runs out of time, the entry will be deleted from the table and the user will be forced to re-query the DNS for a new capability.

C. Honeypot & Captcha

When the user is sent to the honeypot, part of the logging will capture the domain name that the user attempted to query. With this information logged the user will presented with a captcha, or essentially a puzzle that cannot be solved by a non-human user. If the user successfully validates, the information for the original query will be parsed and attempted again.

VIII. Evaluation

A. Empirical Setup

In order to describe the empirical setup in a complete manner we include details on the setup of the system. We do this to show the reader exactly why we choose the metrics of performance that we do.

1) Provisioning: As mentioned above, we will provision as follows: a VM with the DNS and the NAT, a VM with the honeypot, a VM running the asset server and a VM acting both as an attacker and as a legitimate client. We include sample IP addresses for clarity in the examples that follow, where "x" represents our assigned team number.

VM #	Component	IP Address
VM 1	DNS Server	10.4.x.1
VM 1	NAT Router	10.4.x.1
VM 2	Honeypot	10.4.x.2
VM 3	Asset	10.4.x.3
VM 4	Client	10.4.x.4
VM 4	Attacker	10.4.x.4

2) DNS Setup: The DNS Setup consists of configuring BIND9 and configuring it to map the asset and honeypot servers. Specifically, we need to use *netfilter_queue* to capture the DNS response to a user and notify the NAT that the specific public IP address maps to the requested server's secret address. A majority of this setup is very similar to setting up a normal DNS server using BIND9. It is important to note, we run *Aminon* using only the IPv4 protocol. We shall not be testing traffic or requests that use IPv6. Thus we shall not measure *Aminon's* success by its capacity to handle traffic that utilizes IPv6.

3) NAT Setup: The NAT will be deployed to filter and control the incoming traffic through the use of *iptables* on the router machine. The following Python script and *iptables* command set this up cite: [3]. It will modify the DNS responses on the fly to give the user a random public address and map it to the real private address (of the asset server). This step is complicated as we must avoid generating the same public address before its original TTL is over. We will consider the script running the *netfilter_queue* to be part of the NAT. It is what coordinates the NAT and the DNS while they're operating. The *netfilter_queue* will capture packets outgoing from the DNS to a user. It will then update the NAT by writing a rule to map a public IP address to a private IP address using *iptables*. A user will then be able to that correctly resolve the public IP address for the asset server.

Filter traffic through *netfilter_queue*

```
def ip_fluxing(pkt):
    script_start = time()
    if (pkt.ttl + buffer) == 0:
        /* Expired TTL so drop the packet */
        pkt.drop()
        /* Remove the packet IP from the
            used IPs */
        used_ips.delete(pkt.get_ip())
        /* Return to listen for the next
            packet */
        return

    /* Generate a random IP that is not a
        part of used_ips */
    rand_ip = gen_rand_ip()
    if rand_ip in used_ips:
        while rand_ip in used_ips:
          rand_ip = gen_rand_ip()
    /* Extract the private IP from the DNS
        Packet */
    private_ip = ptk.get_dns_answer_ip()
    /* Add the rule to iptables */
    update_iptables(mapping(rand_ip -->
        private_ip))
    /* Edit the DNS Packet on the fly */
    pkt.update_dns_answer_ip(rand_ip)

    script_end = time()
    pkt.ttl += script_end - script_start
    pkt.accept()

nfqueue.run(ip_fluxing)
```

4) Key Concerns and Considerations:

- ***Race Conditions:*** In a DNS system where we are implementing multiple components and require they collaborate, race conditions need to be considered. Race conditions need to be considered when the DNS server is sending back the IP address of a server after a query. We must assure that the correct mapping gets created before the capability is granted and passed back to the client. We want to create some internal structure to control the flow but assure that the client cannot perform certain actions before their query gets handled or the DNS won't translate real traffic to the honeypot before the capability is granted or the mapping created. As detailed above we will be using the kernel's internal queue handled with *netfilter_queue* to overcome these race conditions.
- ***Communication Issues Between Components:*** In a system where multiple parts talk to each other, communication becomes critical to service. If one part of the system goes down, the rest of the system will be at worst unusable and at best lack integrity for communication, both detrimental to the customer and our security goals. We will look at the possibilities of attack during the loss of some key components of the system from the perspective of a non-malicious and malicious user.
- ***System Efficiency:*** Efficiency is a key consideration in this project because we are ultimately aiming to be similar or better in effectiveness to the DNS system currently in place. One backbone point in support of this approach will be the similarity in performance but enhancements in security. This system efficiency will be checked in two specific areas: (1) Client queries for DNS resolution and (2) Client queries to the protected asset server after the capability has been granted. The other key area we will need to check for system efficiency is the rates at which malicious users successfully gain access to the protected asset server but also the rate at which non-malicious clients get wrongly rerouted to the honeypot.

B. Evaluation Methodology

- ***Race Condition Evaluation:*** As explained in the previous section, race conditions will need to be considered when the DNS server is sending back the IP address of a server after a query. These values are reasonable estimates of how many users made a legitimate DNS request and then were still routed to the honeypot server. To do this we will setup logging in both the DNS and the honeypot server. We shall then generate the values over an interval of time during regular activity. We expect this to be relatively a low amount of time because our model only has one client. If the percentage is below 99% (of race conditions resolved), we shall consider the race conditions incorrectly handled. We do not expect there to be any incorrectly routed users due to race conditions. In order to compensate the number of DNS requests intentionally sent to the honeypot during this time, we shall subtract the number of honeypot routed DNS responses from the number of those who escaped from the honeypot and compare these values. This will give us a range of the percent of thus being negatively

affected by race conditions. We shall also investigate the percentage of those who escaped the honeypot. This analysis will occur offline, as follows:

Evaluating Race Condition Avoidance

```
def find_race_condition_range():
   /* loads all the honeypot logs */
   honeypot_logs = load("honeypot_log_")
   dns_logs = load("dns_log_")
    /* loads all the dns logs */
   netfilter_logs = load("netfilter_log_")

   /* get the count of those who escaped
       */
   escaped =
       count(process(honeypot_logs),
       "ESCAPED")
   /* get the count of clients who enter
       the honeypot */
   total_honey =
       count(process(honeypot_logs))
   /* get the count of those
       intentionally sent to the honey pot
       */
   captured = count(process(dns_logs),
       "HONEYPOT")
   /* get the count of dns responses */
   responses =
       count(process(netfilter_logs),
       "RESPONSE")

   min = escaped / responses
   max = max(escaped - captured, 0) /
       responses
   escape_percent = escaped / total_honey

   return (min, max, escape_percent)
```

- ***Communication Issues Between Component Evaluation:*** Communication issues that stem from the client or malicious user's end-host will be considered non-issues as this would only have an effect on the user and no detrimental impact on the DNS infrastructure. The DNS server, NAT, or protected server going offline for any reason would be considered critical to the performance and success of our system working properly, securely, and efficiently. The loss of the protected server will only have detrimental effects on the evaluation of the system performance and cause frustration to the customer but will not affect our infrastructure. We believe the loss of communication with the NAT will render the system useless as none of the other components would be able to use the functionality of any other components. We will evaluate the loss of the DNS server by evaluating whether or not we are able to exploit TTL's that have yet to expire in the NAT.
- ***System Efficiency Evaluation:*** The methodology for system evaluation will not be overtly complex. We will benchmark round-trip time (RTT) for queries for both DNS resolutions and queries for information from the protected server. We will leverage these benchmarks against research and industry results for DNS queries across the Internet. Further considerations are needed when drawing conclusions between the benchmarks due to the fact that our system has a statically placed DNS server that is authoritative versus a real-life implementation which relays through different networks and DNS servers based on the uniqueness and repetitiveness of a certain query. For example, we expect a query to a protected server to be similar since the mapping will already be resolved but we would expect a query that has to be resolved to take longer in a real-life situation where it has to pass through multiple DNS servers and then make it all the way back to a client. We will log attempts by both malicious users and non-malicious users on the success rate they have in reaching the protected server. Success for a client is considered reaching the protected server as opposed to being redirected to the honeypot. Success for the malicious user is considered reaching the protected server versus being automatically redirected into the honeypot. We will log these on a specific testing period as we realize the challenge of collecting data continuously during the life of our system. To determine how long it took for a client to complete a DNS server, receive the request, and then receive a response from the target asset server by using the command line utility 'time.' We are interested in the wall-clock time, and we shall subtract the user and system time from the result because these values are internal and the timing of the DNS request does not affect them.

Timing Client Request

```
time wget asset.com --no-dns-cache
```

C. Metrics and Criteria for Success

The evaluation of *Aminon* will be determined by three metrics. These measures are *functionality*, *deployment ease* and *performance*. Semantically, *functionality* will measures if a component fulfills its purpose, *deployment ease* will measure both the difficulty it is to manage this component, the cost to use it and how feasible it would be to deploy in a larger scheme and *performance* measures how well the system or component does it job. We will be measuring each metric separately, both on the project as a whole and individual components where necessary. Below we define the exact parameters for each metric for each component and the system as a whole.

1) Aminon: The system must provide *availability* to a user. To do this it must defend against the threat model, disallowing attackers that are attempting to disrupt users' connections. Furthermore, it must not false-positively identify a normal user as a threat if given discernible evidence.

- *Functionality*: *Aminon* sets its scope to combat attackers randomly attempting to access IP addresses, although we acknowledge a denial of capabilities type of attack is possible. Therefore, given our previously stated assumptions, we will evaluate the functionality of *Aminon* based on its ability to maintain critical areas. We consider the following cases through the scope of availability:

1) If the NAT or DNS server breaks we lose operational status, meaning the whole system will be unusable, as neither the users nor the malicious users can no longer access the asset server.
2) If the asset server is down, intended functionality is lost completely though our system's intended functionality may have been preserved.
3) If the honeypot breaks, malfunctions, or goes down the core system is functioning and our non-malicious clients can still maintain access to the asset. The honeypot operates on a separate machine thus the functionality of the rest system is preserved and the clients can still access the product server. If the honeypot fails the potentially malicious client trying to access it will simply receive a 404 error, and we just lose the ability to collect honeypot information and dynamically improve our system performance.

- *Performance:* The performance metrics will be measured and compared to researched current DNS statistics. The metric surrounds clients querying the DNS for a capability and an open mapping for communication. Based on some average times we have seen in some of the DNS functionality on our Virtual Machines, we believe the average round-trip latency will measure at about 500 milliseconds to send the query and return with a capability. Anything over 750 milliseconds will be considered to be slow and an issue to be looked into.
- *Deployment ease*: Beyond a matter of self-interest, it is critical that the deployment of this project be easy for network administrators to deploy and maintain but more importantly it doesn't cost a company overhead in DNS performance. We shall consider the metric of *deployment ease* fulfilled if the core modules of the NAT Router and the DNS do not require additional dependencies and setup than what is detailed above, then the project will be considered easy to deploy. it will be considered success and a viable replacement to be considered if it consistently meets these performance goals which would out-perform current DNS.

2) NAT Router:

- *Functionality:* The NAT provides mapping from the public IP Address to the private IP Address of the asset being protected. The functionality of the NAT will be tested by running DNS lookups and then following up the public IPs given out map the client to the asset server through the newly given public IP. It will keep itself maintained based on the TTL's and handle the clearing of unused and expired entries.
- *Performance:* The NAT will provide the most added overhead to the system by having to maintain all of the mappings to their hidden private IPs. We expect this to take very minimal extra amounts of time in the initial mapping but negligible extra time in handling requests through the mapping. To compare performance of our custom NAT with a stock NAT, we would run tests to time a public IP lookup on the original NAT and compare the same with the public IP lookup on our custom NAT.
- *Deployment Ease:* In our case, the DNS and the NAT were running on the same virtual machine as the DNS server. We used a Python script to incorporate the necessary libraries together to deploy the functionality. A few metrics for measuring ease of deployment include: time to configure, number of lines to be edited in the configuration files, number of commands to run to setup the system and the ease of using our Python script to setup the NAT server.

3) DNS:

- *Functionality:* The DNS server will be tested for the accuracy of its results as well as its speed when returning with answers to queries.
- *Performance:* In practice, we could expect the custom DNS to introduce minimal delays (order of milliseconds) to DNS lookups due to the previously mentioned overhead for mapping IPs before opening the connection. To compare performance, we would time DNS lookups on a stock DNS server and compare timings for the same lookups on our custom DNS server. This metric is factored into the measurements of the total request time.
- *Deployment Ease:* The DNS is also going to be running on a machine in the network that has the server that is being protected. The DNS will be listening in for the authoritative DNS Queries related to the hostname of the server being protected. The DNS relies on the *BIND9* public DNS server. This server must be easy to setup and configure. Metrics that indicate ease of deployment would include: number of lines to be edited in configuration files, number of commands to run to setup the system, and ease of using any shell scripts bundled with the project to facilitate the setup of the DNS Server.

4) Attacker:

- *Functionality:* The attacker will be evaluated for its ability to attempt to break our system in as many ways possible. The attacker should attempt to break in from all possible avenues and exhaust all the options available. One possible metric that could be used to measure the functionality of the attacker is to count the number of seconds before which the honeypot server stops responding in a DoS Attack on the network.
- *Deployment Ease:* We aim to make sure that the attacker is a potential passive threat that will actually test the system through scripts such as the following scanning script below:

Attacker Scanning Script

```
while(1):
   for ip in public_ip_space:
     wget ip --no-dns-cache
     if index.html contains asset server
         secret line:
       log("Scan successful " + ip)
      else:
       log("Scan unsuccessful")
```

We will also use a tool we developed for Denial of Service attack. We have formatted the script so that if

it is able to access an asset server in our network, the retrieved page is logged and the success is also logged. We can then rely on this output file to compare our total trials to the number of successes to evaluate a percentage or proportion. We believe the successful attempts to be extremely low, with very few 'lucky' guesses being routed to the asset server. If it is over 1% we have implemented something incorrectly in the system. We plan on testing the success of the attacker versus the TTL of the DNS capability and believe the results will follow an inverse relational pattern.

5) Honey Pot and Captcha:

- *Functionality:* The honeypot has one role: to test if the user that is attempting to gain access to the protected resource is legitimate and not a bot. To verify the legitimacy of the user we present the user with a captcha text to be filled out. We will run tests and record the success rate of a user being redirected out of the honeypot for correct responses to the captcha.
- *Performance:* The honeypot server will be subject to stress tests, for example: high volumes of traffic (potentially running a DoS attack from the attacker's end). The multi-threaded nature of the honeypot server should be able to handle multiple simultaneous requests. However, due to limited memory resources on the virtual machine running the honeypot, there would be a point where the web server would stop responding due to the inability to spawn more threads to listen to web requests. This will be the breaking point for the honeypot and these results will be noted. Anyone who abuses the DNS and tries to get to the asset server strictly through querying the IP will be routed to the honeypot server and served a captcha. We aim to get at least 99% of the malicious queries routed to the honeypot. We will provide text-based questions that cannot be parsed for an answer and thus we will **not** measure the percentage of the malicious users that were able to escape. However, we believe that a human user could answer the question and get out of the honeypot 100% of the time.
- *Deployment Ease:* The honeypot server would have to be implemented on the network that has to be protected by the attacker or malicious clients. The honeypot **could** be running on the same machine as the DNS and the NAT. However, separating the two and running them on different machines allows for easier troubleshooting of issues. The honeypot will be a Python script that runs a web server on a random port (in our case because we are running it on a different machine, it will serve pages on the default port 80). The ease of deployment will be measured with the time it takes for the server to start after the command to fire up the server is issued and the number of commands to start up the server.

IX. Results

In this section we present the data and trends we observed when testing the *Aminon* implementation against the metrics and attacks outlined in the Evaluation Section. Once presented, we attempt to justify the results we collected and compare them to the results we hypothesized in the Evaluation metrics.

A. Summary

In this section we will present our findings for the four main areas of measurement outlined in the evaluation section:

1) The average duration taken for a client to query for a capability and be answered with or without one. This includes the average time it takes to use the new mapping in the NAT to access and retrieve a webpage from the secured asset server.
2) The rate of success interpreting the number of non-malicious queries made that were successfully routed versus the number of non-malicious queries that couldn't be distinguished or were wrongly distinguished and were routed to the honeypot.
3) The success rate to which we block malicious queries and direct them to the honeypot versus the number of malicious queries that make it to the protected asset server and retrieve a webpage. We will also compare this metric to the metric found in 2).

Capability Query Results Timing		
Query Version	Capability Granted Average Percent (%)	Average Query Duration (ms)
Regular DNS (*wget*)	100	46.51
ping (*nslookup*)	100	50.94
dig	100	50.50
Average for all the commands	100	49.32

The above table begins the presentation of our results for non-malicious queries to *asset.com*. We attempted 100 queries each with each of the three commands: *wget, nslookup, and dig*. This simulated 3 different access strategies for non-malicious users. In all of the trials we were able to correctly grant the user the capability and return in the reported average times in the third column. Across the 3 different lookup schemes we were able to complete the query and grant the capability on average in just under 50 milliseconds.

Average Time to Access Protected Server			
Number of Queries	Fastest Time (in ms)	Average Time (in ms)	Slowest Time (in ms)
300	41.99	46.51	72.00

This above table shows the variation we observed when timing the queries which were all run with "no cache" options when supported by the outlined commands. The slowest query was under 75 milliseconds which was significantly under what we believed we would see. However, we attribute this

to having the DNS routed through the authoritative server. We expected around 500 milliseconds based on averages from research and concluded previously that anything over 750 milliseconds would be a failure. When stepping back and re-examining the steps taken to resolve a query in a regular network versus our closed network, we conclude anything over 200 milliseconds would be considered a failure. However, when using the same virtual box environment to query popular domain names we saw that our implementation was slower on average by around 15 milliseconds. We attribute this differential to the overhead in granting the capability and holding the response until the NAT is updated accordingly. In scaling this system, we believe that if it continued to have an authoritative server and only one NAT that the query times would remain similar to our results. However, any addition to those components would result in more common DNS lookup times which we drew conclusions from in the evaluation section.

Efficiency with Non-Malicious Requests				
Total Number of Queries	Successful Queries	Percent Success (%)	Failed Queries (honey-pot)	Percent Failure (%)
300	300	100	0	0

The efficiency for requests by non-malicious users as seen in our collected results was a full 100%. We had earlier stated that anything under 99% would be considered a failure for our system. Therefore, based off this statistic, we consider the implementation to be a success in this regard.

Efficiency with Malicious Requests				
Total Number of Queries	Blocked Queries	Percent Success (%)	Failed Queries (accessed asset)	Percent Failure (%)
300	300	100	0	0

The efficiency for blocking malicious queries was also a full 100%. In the 3 trials of 100 queries each with the three commands, the system was able to identify the query as potentially malicious and route it to the honeypot. We came to this result by running the malicious queries and then parsing the returned index files for the secret honeypot phrase in the html header. We previously concluded that anything over 99% would be considered a success and therefore we consider the system to be successful in this regard.

Aminon also dealt well with the DNS and NAT Race Condition. In using *wget* with the **–no-dns-cache** option, *wget* would initially perform a DNS request followed by the actual query to the domain name requested. 100% of the time, *wget* returned with a web page from the correctly defined destination of the request, meaning the NAT was updated before the DNS response was sent out. This implies that the race condition was successfully handled by *Aminon* in 100% of the queries.

B. System Functionality

1) Client's Query Time: The average time for a client's query, as noted above, is 50.5 milliseconds. The following table present the results of querying popular domains names with the *dig* command as compared to our implementation. Also presented in the graphs following in **Figure 2**, **Figure 3**, and **Figure 4**, outline the variation of query times by the query command we ran. This data is collected from 3 trials of 100 executions for each command. *wget* was seen to be the overall quickest command but only by about 4 milliseconds as seen by the average time in the original table of averages. Also presented in **Figure 5** is the average DNS query time for popular domains. As we presented earlier in the section, our implementation was generally around 15 milliseconds slower on average, with the fastest time being significantly slower than the fastest time for the other domains.

Average Times of Popular Domains			
Destination Domain (Number of Queries)	Fastest Time (in ms)	Average Time (in ms)	Slowest Time (in ms)
google.com (300)	23	36.17	50
wpi.edu (300)	21	31.3	94
8.8.8.8 (300)	64	34.46	101
asset.com (300)	45.00	50.5	66.99

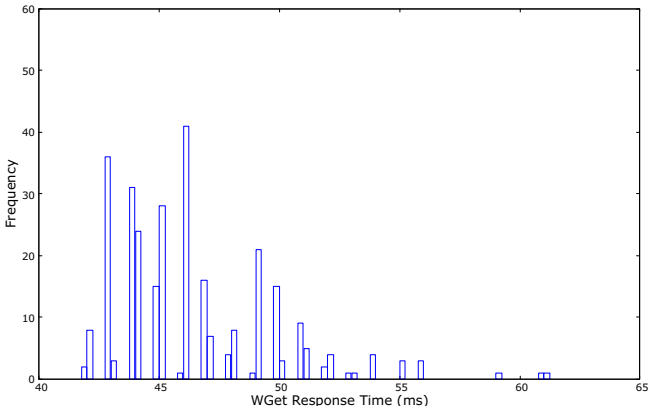

Fig. 2: Aminon *wget* times

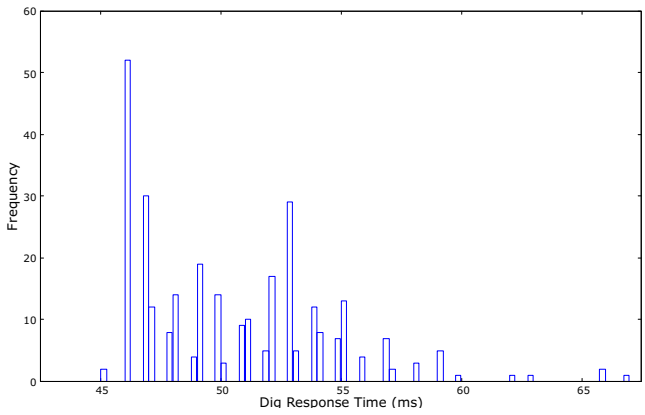

Fig. 3: Aminon *dig* times

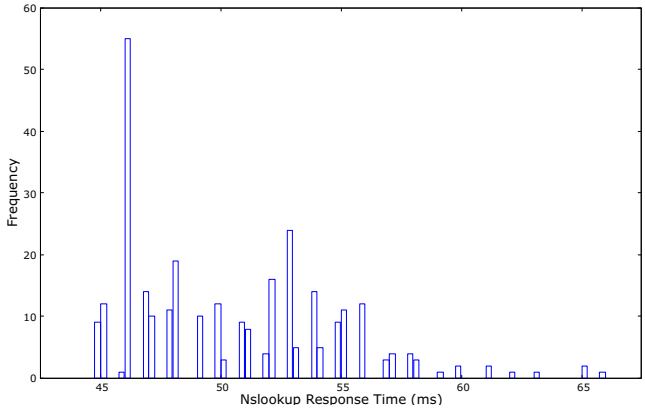

Fig. 4: Aminon *nslookup* times

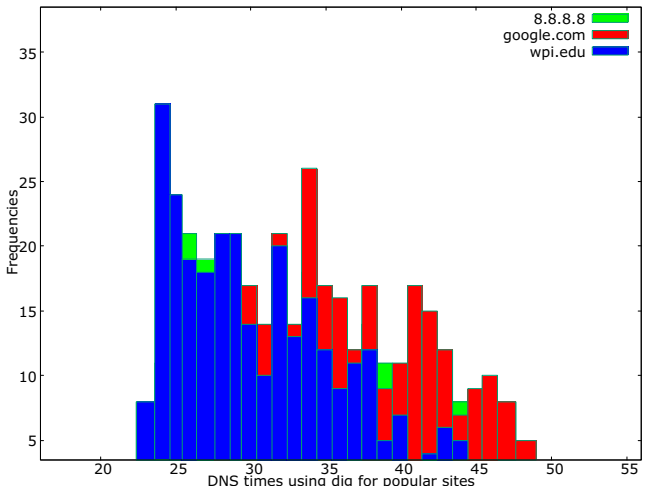

Fig. 5: DNS Lookup times for popular times *(via dig)*

In the above graph, blue represents the domain *wpi.edu*, red the domain *google.com*, and green the google DNS server located at IP *8.8.8.8*.

As seen in the above table and subsequent graphs, our fastest times were slower than the normal DNS times for popular domains but very competitive in the slowest range. In the average case, the times were only slightly slower than the popular domains. and we believe this small increase is not enough to make a noticeable difference in the client's experience, especially since it is only an increase of 15 milliseconds when the worst case scenarios for some of the domains push 50 milliseconds over the average. Therefore, we are considering the system successful in this regards especially from the standpoint of being a viable alternative with the already saved costs of sustaining the current infrastructure.

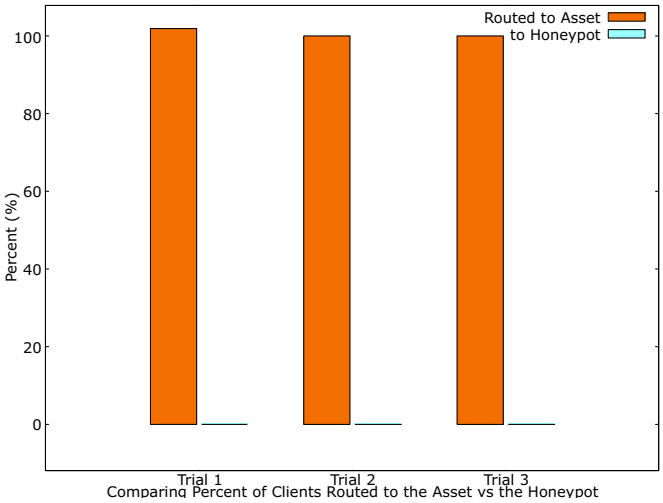

Fig. 6: Comparing percent of non-malicious users reaching the asset server by performing a DNS lookup on *asset.com*. 100% reach the asset.

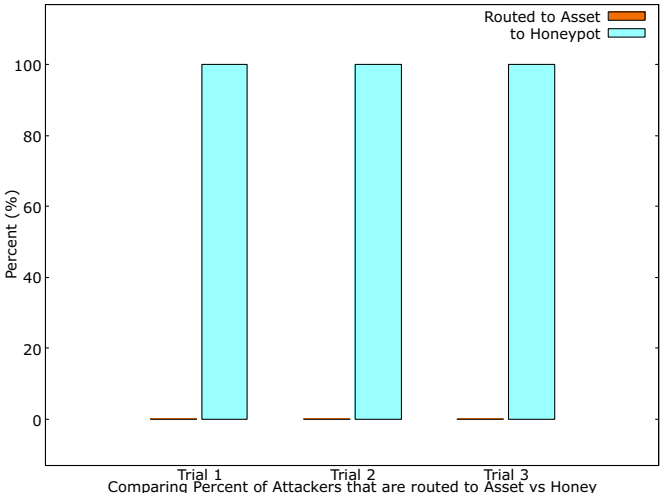

Fig. 7: Comparing percent of IP-Scanning Probes being routed to the honeypot vs the asset Server by searching through the public ip space. 100% are routed to the honeypot.

2) Attacks Deflected to the Honeypot: First we show that when accessing the asset server by querying the DNS server (as a non-malicious users), that all of the requests were able to be resolved and successfully access the asset server. Second, we compare that value to the percent of attacks which reach the asset through randomized (public) IP Space scanning. Each bar

in the above graphs represents a trial of 100 runs. *Aminon* was able to correctly recognize the 100% of the IP scanning probes and route them to the honeypot server. All of these probes were tracked as shown in **Figure 8**. As mentioned earlier we did this through collecting the index files and searching for secret strings we placed in the honeypot. The automated attackers who were redirected to the honeypot server would be stuck as the automated scanner would not be able to answer one of the human related questions given to them, such as "What numbers does the string **n!yn** resemble? Write the word for the number." (Answer: nine). We previously concluded that we believe the average for non-malicious users to consistently reach the asset server to be 99% or greater, and the average for malicious scanning users to consistently be blocked and routed to the honeypot to be 99% or greater. Therefore, with the given findings, we consider the system to be a success in these metrics.

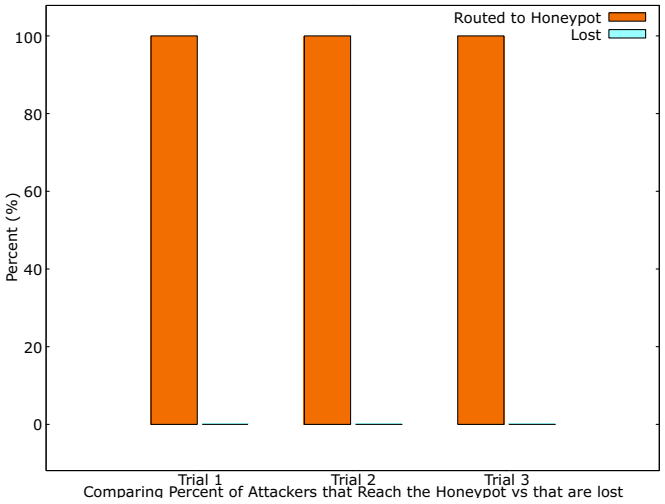

Fig. 8: Comparing percent of attackers that were sent to the honey pot and logged vs those that were lost. 100% of attackers were recorded and tracked in the honeypot server.

3) Real Users Routed to Honeypot: We want to maximize the number of real users getting access to the asset, and minimizing the number of attackers who do. We believe with the findings for correctly discriminating between malicious and non-malicious, and the ability of users to escape from the honeypot that our system consistently accomplishes this goal. Our system was a success as 0% of the real users were sent to the honeypot, this can be seen in **Figure 6**.

*4) **D**enial **o**f **S**ervice (DoS) and **D**enial **o**f **C**apability (DoC) attacks:* We developed tools to simulate both DoS and DoC attacks. The DoS script continually created new threads that requested resources from the server. We did not successfully accomplish the goal of DoS-ing the web server as the traffic was redirected to the honeypot, where a team member on a separate VM was still able to retrieve resources from the server while the DoS script was running. We believe this failure resulted from not having the resources to spawn enough requests with the supplied VMs before either the server or the attacker machine crashed itself. We attempted to use tools as well such as *slowloris* and *goldeneye*, but neither tool resulted in a desirable result. We believe a similar failure occurred with the DoC script. In the DoC script we continually deployed new threads (up to a certain limit of 200 threads) to request capabilities through the DNS server. The attack continually resulted in the attacker's VM crashing, not allowing us to reach the end result of the attack. We believe our system would be resilient to these two kinds of attacks, however, because of the successes we had identifying malicious from non-malicious behavior and successfully routing it to the appropriate destination in our testing of the system.

X. Conclusions

In this report we presented *Aminon*, a system leveraging the currently implemented DNS infrastructure to subdue the ongoing problem of unwanted traffic on the Internet. Through taking advantage of non-malicious users' tendency to rely on DNS resolution versus directly querying with IP addresses, we are able to consistently provide service to clients while stopping attackers from using automated attacks. As seen in our testing of the implementation, the system was able to discriminate between malicious and non-malicious traffic and direct the former to a honeypot server and the latter to their requested asset in an efficient and quick manner. We were able to provide service to clients in a time similar to that of the current DNS infrastructure but with the addition of securing web servers to provide uninterrupted service. We concede this implementation does open new attack vectors that need to be secured, but it defends against some of the most prevalent attacks on the web such as IP Scanning and Denial of Service attacks.

Although *Aminon* provides a usable implementation there are many things someone could implement in future work. First would be a robust testing system. Either through an automated framework or through unit and scale testing, the system could be put under more and longer amounts of stress to further understand its performance metrics. Another thing that could be added is a more secure honeypot server that uses ideas such as picture captchas or password verification to get out of it.

XI. Improvements

A. CHANGE LOG

Design Document Changes:

1) Complete overhaul of the Abstract Section and a new Introduction
2) Added IP spoofing to the Threat Model
3) Clarified the timing of updating the NAT table
4) Lengthened explanation of thr Captcha Functionality
5) Cleaned up code pieces for readability and simplicity

Evaluation Document Changes:

1) Moved some details we had that were better suited for design document
2) Clarified points in the criteria for success section
3) Lengthened details of attacks to our system along with more specific details

4) Removed installation code snippets and some extraneous configuration snippets as they added
5) Fixed a few typos
6) Cleaned up code pieces for readability and simplicity

Results Document Changes:

1) Replaced all of the placeholder data with real data and added justifications of the collected data
2) Added in quantities of tests run for each metric
3) Cleaned up the graphs and figures
4) Added clarification to graphs and tables
5) Clarified the testing metrics for what constitutes a success, failure, etc.
6) Completely new Conclusions Section

REFERENCES

[1] https://www.gnu.org/software/wget/manual/wget.html
[2] https://www.digitalocean.com/community/tutorials/how-to-configure-bind-as-a-private-network-dns-server-on-ubuntu-16-04
[3] https://github.com/kti/python-netfilterqueue