Accurately Measuring Global Risk of Amplification Attacks using AmpMap

Soo-Jin Moon, Yucheng Yin, Rahul Anand Sharma,
Yifei Yuan, Jonathan M. Spring, Vyas Sekar

October 13, 2020

CMU-CyLab-19-004

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Accurately Measuring Global Risk of Amplification Attacks using AmpMap

Soo-Jin Moon[†], Yucheng Yin[†], Rahul Anand Sharma[†], Yifei Yuan[§*], Jonathan M. Spring[◇], Vyas Sekar[†]

[†]*Carnegie Mellon University,* [§]*Alibaba Group,* [◇]*CERT/CC®, SEI, Carnegie Mellon University*

Abstract

Many recent DDoS attacks rely on amplification, where an attacker induces public servers to generate a large volume of network traffic to a victim. In this paper, we argue for a low-footprint Internet health monitoring service that can systematically and continuously quantify this risk to inform mitigation efforts. Unfortunately, the problem is challenging because amplification is a complex function of query (header) values and server instances. As such, existing techniques that enumerate the total number of servers or focus on a specific amplification-inducing query are fundamentally imprecise. In designing AmpMap, we leverage key structural insights to develop an efficient approach that searches across the space of protocol headers and servers. Using AmpMap, we scanned thousands of servers for 6 UDP-based protocols. We find that relying on prior recommendations to block or rate-limit specific queries still leaves open substantial residual risk as they miss many other amplification-inducing query patterns. We also observe significant variability across servers and protocols, and thus prior approaches that rely on server census can substantially misestimate amplification risk.

1 Introduction

Many recent high-profile Distributed Denial-of-Service (DDoS) attacks rely on *amplification* [54, 57]. In an amplification attack, an attacker spoofs the victim's source IP address and sends queries to a public server (e.g., DNS, NTP, Memcached), which in turn sends large responses to the victim. If a source IP address can be spoofed, any stateless protocols in which the response is larger than the query can be abused. While there are various best practices to mitigate this situation (e.g., [1–3]) given that spoofing is possible, they are unevenly applied. Spoofing the victim's IP may be avoidable in a future Internet (e.g., [26]), but it continues to be possible from a large number of ISPs [11, 23]. Finally, there continue to be many public-facing servers that can be exploited for amplification [57]; many servers do not apply best-practice mitigations (e.g., rate limiting, restricting access).

As networks evolve and server deployments change, the potential for amplification attacks changes over time. For instance, new avenues for amplification emerge (e.g., botnet, gaming protocols), and unexpected vectors for known protocols are discovered [16]. In light of the continued threat of amplification, we argue that we need an Internet-scale monitoring service that can systematically and continuously measure the empirical risk of amplification [7, 13]. We envision a service that periodically maps each server to query patterns yielding high amplification and quantifies these amplification factors (AF). Such a framework can serve as an empirical foundation for cyber-risk quantification that many have argued for [5, 10]. Furthermore, this framework can inform remediation efforts such as throttling servers, generating signatures, informing protocol changes, and provisioning defenses.

At first glance, it seems that we can use or extend existing scanning services that look for and enumerate open/public servers for different protocols (e.g., Censys [34], ZMap [35], and `openresolver` [9] monitor open DNS resolvers, and `shadowserver` [19] reports on open CharGen, LDAP, QOTD, and SNMP servers, among others). For instance, we can multiply the number of open servers with previously reported amplification factors (AF) [5, 57]. We can also extend these scanning services to probe servers using a set of known query patterns (e.g., send ANY requests to DNS servers) to account for per-server factors (rather than using a single global amplification factor for all servers). Unfortunately, these have fundamental shortcomings (§2.2). These solutions assume that the amplification that servers yield is homogeneous or that they share an identical set of query patterns. In reality, we see significant and unpredictable variability in amplification across servers (including within servers running the same software versions) and query patterns that yield amplification. Thus, these approaches are inaccurate for estimating the empirical risk and for informing remediation efforts.

At the other extreme, we can envision a brute-force approach of sending all possible protocol-compliant queries to

*Contributions by Yifei Yuan were made during the time he was a postdoctoral researcher at Carnegie Mellon University.

	Known pattern	AmpMap-discovered patterns	
		new pattern	polymorphic variants
DNS	EDNS:0, `ANY` [1], `TXT` [18] lookups	EDNS $\neq$ 0, `LOC`, `SRV`, `URI` lookups ...	rd:0 (off) DNSSEC:0 (off) EDNS payload<512 ...
NTP	`monlist` [2,57]	`if stats` `if reload` `get restrict` `peer list` ...	None
SNMP v2	*GetBulk* request [3,57]	*GetNext* request *Get* request	Vary an object identifier (OID); Vary the # of OIDs
Chargen	Character generation request [57]	None	None
Memcached	Stats command [3]	None	None
SSDP	SEARCH request [3,57]	None	`ssdp:all` `upnp:rootdevice` ...

Table 1: Summarizing known, unforeseen, and polymorphic query patterns found using AmpMap

servers for each protocol. Unfortunately, the search space of possible queries is large (e.g., NTP has multiple 32-bit fields). We can also consider simple fuzzing or existing heuristic-based optimization techniques but they all have fundamental limitations as the relationship between the packet field values and amplification can be quite complex. This highlights a fundamental tension between the overhead of such an amplification-monitoring service and its utility.

In this paper, we present AmpMap, a framework for measuring the risk of amplification with a low network footprint that accounts for both the server- and query-specific variability. Our approach builds on key structural insights. First, we observe that distinct amplification-inducing query patterns overlap in terms of values in protocol fields. This locality structure suggests that if we find one such pattern, we can potentially uncover other related patterns. Second, we observe that large fields (e.g., 16 or 32 bit) either do not affect amplification (e.g., timestamp for NTP), or when they do, have contiguous structure (e.g., EDNS payload for DNS). This structure suggests that we can use smart sampling strategies to efficiently explore the search space of large fields. Finally, even though protocol server implementations are diverse, they share some similarities. This helps us further reduce overhead and improve fidelity by sharing insights across servers.[1]

Findings: We implemented AmpMap, validated our parameter settings in lab settings, and ran real-world measurements. Our key findings (§5) are :

- *Uncovering new patterns and polymorphic variants*: We discovered new patterns and polymorphic variants (from known ones) in addition to confirming findings from prior work (e.g., *GetBulk* for SNMP [3], `ANY` or `TXT` lookups for DNS [3, 57, 62]). Table 1 summarizes our findings. For DNS, we also uncover multiple patterns (e.g., `URI`, `SRV`, `CNAME` lookups) that collectively incur 21.9 $\times$ more risk than a popular-known pattern (`ANY` lookup). Specifically, while some of DNS patterns have been pointed by (mostly) the operational community (e.g., `A`, `RRSIG` [58, 62, 64]), many have not been documented to the best of our knowledge. For NTP, apart from the `monlist` request, we discover `get restrict` and `if stats` can too also incur higher than 500$\times$ amplification factor (AF). For SNMP, apart from *GetBulk* [3,57], *GetNext* requests can incur amplification up to a few hundred! We also discover polymorphic variants due to server diversity. For *GetBulk* request, SNMP servers can incur *magnitudes* higher amplification with requesting for certain object identifiers (OIDs) and querying the right number of OIDs.
- *Variability across servers and protocols*: We observe significant variability with the amplification that each server can yield; e.g., the amplification factor (AF) can vary between 0 to 1300 for NTP. This confirms we cannot assess amplification risk by looking at mega-amplifiers or simply counting the number of servers. We also observe substantial variability in the AF distribution across protocols; e.g., 60.4% of Chargen servers can yield AF above 100 but only 0.02% of servers for DNS. Such variability across multiple dimensions calls for the need to do periodic measurements rather than one-time analysis.
- *Empirical risk quantification*: By analyzing our measurement data, we unfortunately find that just disabling the few known patterns (Table 1) is far from enough; e.g., blocking EDNS0 and `ANY` or `TXT` lookups for DNS still leaves 17.9$\times$ the residual risk from "other" patterns (Table 6). Further, using an additive risk metric (§2), we highlight the imprecision of the risk estimated by prior work. Even if we focus on the known patterns (e.g., *GetBulk* for SNMP), existing techniques underestimate SNMP risk by 3.5$\times$ and overestimate Memcached risk by 5.6K$\times$ and DNS by 1.9$\times$. If we consider new patterns, then the inaccuracy gets worse; e.g., DNS risk is underestimated by 11.9$\times$.

Ethics and Disclosure: We carefully adhered to the ethical principles in running our measurements (§6.1). We have also disclosed the newly discovered patterns to relevant stakeholders such as CERT, vendors, and IP address owners (§6.2). We also discuss countermeasures in light of our findings (§8).

2 Background and motivation

We start with background on amplification attacks. We then motivate the need for empirically measuring amplification risk and discuss why strawman solutions are insufficient.

Primer on amplification: In an amplification attack (Figure 1), the attacker spoofs a victim's source IP and sends a small query/request (e.g., 60 bytes) to one or more pub-

[1] While we acknowledge that these insights may not be universal for all protocols, these hold in practice for many protocols that have been popular targets.

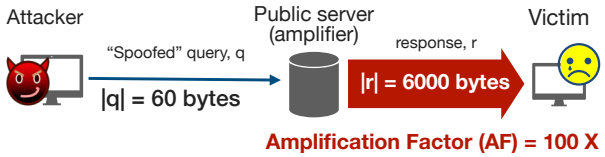


Figure 1: Primer on amplification attack and amplification factor (AF)

lic servers that act as *amplifiers*. Amplifiers send large responses to the victim. The amplification factor (AF) is the ratio of the query and response sizes, e.g., $\frac{|r|}{|q|} = 100$ in Figure 1. AF is also referred to as BAF (i.e., bandwidth AF) in prior work [5, 57]. (We do not report packet amplification factors or PAF [57] for brevity.[2]) Amplification attacks are well known [54] and have been exploited at scale [16, 21, 22]. For example, one of the **query patterns** that induce high amplification for DNS is ⟨ EDNS:0, EDNS payload:(1000,65535), record type:`ANY` ⋯⟩. Here, EDNS is set to version 0, allowing a DNS server to use the non-default payload size and send large responses (default value is 512-bytes). The EDNS payload is set to greater than 1K to overwrite the default 512-bytes, and record type is set to `ANY` to look up all records for a given domain.

2.1 Motivating use cases

We summarize two motivating use cases as argued by prior academic and policy efforts (e.g., [5, 10, 57]). For both use cases, there are two relevant aspects for each server/amplifier: (1) *which query patterns* cause large amplification, and (2) *how much amplification* each query pattern induces.

U1) Assessing cyber risk: Network operators need to know whether, and by how much, their deployments are susceptible to amplification. Policy makers and Internet security experts need a risk assessment to focus their remediation efforts on the highest priority risk. Given a query pattern, p, for a protocol, *Proto*, and a set of servers, S, we define a simple additive risk metric as follows:

$$RiskMetric(p,S) = \sum_{s_i \in S} \text{AF}(s_i, p) \quad (1)$$

Then, given a set of patterns, P, the total risk then is the summation of the risk for each pattern, $p \in P$. Even though this does not consider other factors [5] (e.g., outbound link capacity), it is an instructive metric to quantify risk.

U2) Inform defense efforts: Operators need to know which query patterns induce high amplification to take appropriate defenses (e.g., block or throttle responses). Similarly, protocol designers need to know these patterns to (1) guide the design of future protocols, and (2) assess whether particular remediation (e.g., disabling a feature) can reduce the risk. Lastly, ISPs need to know the degree to which servers are susceptible to amplification to inform capacity provisioning for defenses. For this, the per-pattern risk can also help prioritize the remediation efforts to focus on the largest threats first.

2.2 A case for a measurement service

Given these use cases, we can consider some seemingly natural strategies derived (or extended) from prior work in amplification analysis (e.g., [5, 32, 57]):

- *S1) Scan for open servers*: Using a count of the number of open servers, we can multiply this number by a fixed, known AF (e.g., 556 for NTP [24]). For instance, if there are 1M open NTP servers, this approach would multiply 1M by 556 AF; for a 50 bytes request, this translates to 27.8 billion bytes. Such information can be used for risk quantification (U1) and for informing network operators of their servers (U2) akin to existing efforts (e.g., [5]).
- *S2) Probe servers using fixed patterns*: S1 assumes that servers have identical risk and does not account for multiple patterns. A more advanced strategy is to probe servers using previously known patterns and record their AFs (e.g., DNS [61], NTP [32]). Then, we can use this to assess risk (U1) and construct signatures (U2). However, there can be different options for choosing which patterns to probe (e.g., taking the known patterns, taking the top-K patterns from random sampling).
- *S3) Customize S2 for different server software*: S2 did not account for the variability of query patterns across servers. If servers with the same software setup have similar patterns, then we can run (S2) once for each software setup (e.g., Bind 9.3, Dnsmasq 2.76). That way, we can reduce the number of probes we send.

To understand if these strategies are effective, we run a small-scale measurement study using DNS as an example. We use DNS as its amplification properties are seemingly well understood [24, 57]. We identify a set of 172 queries based on three fields (record type, EDNS, recursion desired, or rd for short) that are known to affect amplification [1, 3, 57].[3] (As we will see later, these three fields do not represent the full set of fields that affect amplification. Rather, we use this as an illustrative set of query patterns to highlight why these strategies are imprecise.) Then, we pick a random sample of 1K DNS servers from Censys [34], send each of the 172 queries, and record the AF per query. We also obtained the version string (if available) for each server using Nmap.

In this dataset, we observe 94 unique patterns that incur $\geq \delta$ AF, where δ=10, with a total risk of 125.8K AF (using Eq. 1); if these servers are connected to a mere 10 Mbit/sec connection, 125.8K translates to 918 Gbps across 1K servers.[4] Using this "ground truth", we evaluate the above strategies using two metrics: (1) the risk estimation accuracy (for U1); and (2) the number of missed query patterns (for U2).

[2] PAF is the the number of IP packets that an amplifier sends for a request.

[3] We generated 172 queries using combinations of 43 values of record type={A, NS, CNAME, ⋯}, EDNS={0, 1}, and rd={0, 1}

[4] 60 bytes/query × 128.5 avg AF / server × 1K servers × 8 bits/byte × 14,880 query/sec (using 10 Mbps and a frame size of 84 bytes)

Strategies		% Error in Risk (U1)	# of Missed Patterns (U2)
S1	Scaling by number of servers	4.5× ↓	N/A
S2	Using known patterns	5.7× ↓	90 (out of 94)
	Top-K from random samples	20× ↓	86 (out of 94)
	Top-K from ground-truth data	3.6 × ↓	84 (out of 94)

Table 2: Effectiveness of S1 and S2 in enabling use cases

Table 2 summarizes these metrics for S1 and S2. For S1 of multiplying the number of servers by a known AF factor, we use an amplification factor of 28, as reported earlier [1]. For S2, we considered three possible instantiations: (1) using known query patterns from prior works (EDNS:0 and record type set to `ANY` or `TXT` [1, 62]), (2) using the top-10 queries across servers w.r.t. the AF values after randomly sampling 20% of the possible values of three fields space; and (3) using the global top-10 patterns from the ground-truth data. Note that (2) and (3) are extremely generous; in practice, we do not know the global top-10 a priori, and the actual space of queries is much larger than just 172 queries. We see that S1 of scaling server count under-estimates the risk by 4.5×. Depending on the scaling factor, the risk may also be significantly over-estimated. S2 also under-estimates the risk (U1). We also see that S2 misses many query patterns (U2).

We also observe that this aggregate estimation error across 1K servers translates to large percentages (%) of residual risk for each server (if we had used S2). If we consider a cumulative distributive function (CDF) of the % of the residual risk for each server, 50% of the servers would have: (1) $\geq$ 68% residual risk (if we had blocked the top-10 patterns from the ground-truth, which is infeasible in practice), (2) $\geq$ 72% residual risk (if we had blocked only the known patterns), and (3) $\geq$ 82% residual risk (if we had taken top-10 patterns after random sampling the header space). The trend does not really get better, even if we had used other top-Ks (e.g., 20).

Finally, Table 3 shows the ineffectiveness of S3 for the top-5 version (ranked by the number of servers that have at least one query that induces AF $\geq \delta$ in the dataset). Here, we define that servers have identical software setup if they share the same vendor and a major version.

	% Error in Risk Estimation for U2; (# of Missed Patterns / # of Total Patterns) for U2				
	Microsoft 6.1	**Dnsmasq 2.52**	**Dnsmasq 2.40**	**Dnsmasq 2.76**	**Bind 9.9**
Using known patterns	14.4× ↓ (76/80)	2.7× ↓ (27/31)	6× ↓ (38/42)	3.8× ↓ (44/48)	8.8× ↓ (72/76)
Top-K from random samples	8.7 × ↓ (70/80)	3.6 × ↓ (27/31)	44.2 × ↓ (41/42)	31.6 × ↓ (45/48)	7 × ↓ (66/76)
Top-K from groundtruth	4.5× ↓ (70/80)	1.2× ↓ (21/31)	3.8× ↓ (31/42)	1.7× ↓ (38/48)	6× ↓ (66/76)

Table 3: Effectiveness of S3 that does per-version analysis

To understand why these strategies are inaccurate, we analyzed this data further. To explain our analysis, we define some terms. Given a server, s_i, let Q_i be the set of queries that incur AF $\geq \delta$; Q_i is the set of queries that elicit large responses. Given n servers, let Q be the union of $Q_1 \cdots Q_n$; Q is the union of all amplification-inducing queries.

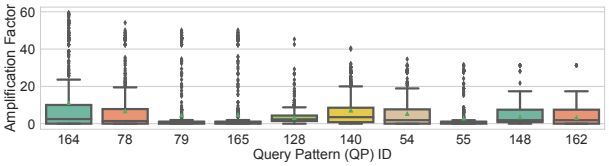


Figure 2: Diversity of AF given a query across servers

Variability in magnitude across servers: Figure 2 shows the distribution of the AF value across servers. (Due to space, we only show this for 10 queries that induce the highest AF if sorted by the AF across our dataset.) For a given q, the standard deviation ranges from 3.9 to 17. Looking beyond the global top-10 queries, if we consider a maximum AF for each server (across all 172 queries), there is significant variability with a standard deviation of 16.7. This trend also holds for servers sharing the same software versions (not shown).

Variability in query patterns across servers: If only a small subset of patterns induce amplification on *all* servers (i.e., Q_i are identical), then S2 and S3 would have been sufficient. To this end, we analyze the similarity (or lack thereof) of query patterns across servers in two ways. Let TopK(Q_i) denote a set of Top-K queries when Q_i is sorted by the AF value. Then, we analyze: (1) How similar are high-amplification query patterns between every pair of servers (i.e., TopK(Q_i) from TopK(Q_j))? (2) How similar is a server-specific query pattern set, TopK(Q_i), to the global set, TopK(Q)? We compare the top-K queries where K=10. Note that we are not just looking at the maximum query (K=1) as we want to consider multiple patterns. We observe the same trend holds for varying Ks such as 5, 20 (not shown).

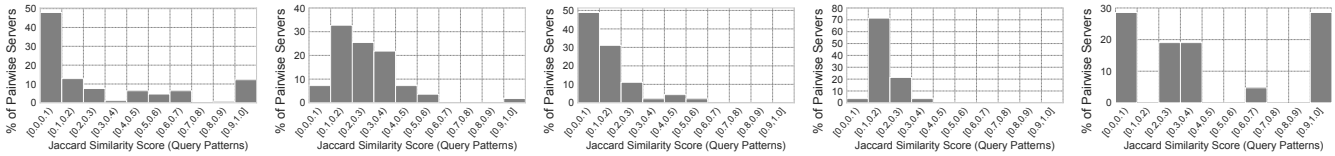
If we look at the histogram of similarity score when K is 10, more than 60% of server pairs have low similarity scores equal or below 0.2, and only 4% of server pairs have above 0.8 similarity scores. This trend is also similar for servers with identical software (Figure 3). For example, more than 45% of Microsoft 6.1 servers have similarity scores ≤ 0.1. For the question (2), compared to the global TopK(Q), we find that more than 70% of servers' TopK(Q_i) has ≤ 0.2 similarity scores w.r.t. the global TopK(Q).

Taken together, these results suggest that we cannot attribute the homogeneous risk per pattern and across servers. Furthermore, we cannot just extrapolate the risk from one server instance (or one per software version) for our use cases. Given this empirical variability across servers, query patterns, and the AF values, we argue that we need an active measurement framework to quantify the risk and inform defenses for amplification attacks.

3 AmpMap Problem Overview

Having made a case for a measurement service, we formulate the goals for such a service we call AmpMap. Then, we discuss the challenges in realizing such a service.

Formulation: We consider S servers implementing a proto-



(a) Microsoft 6.1 (b) Dnsmasq 2.52 (c) Dnsmasq 2.40 (d) Dnsmasq 2.76 (e) Bind 9.9

Figure 3: Histogram showing the Jaccard similarity scores between Top-10 query patterns of pairwise servers

col, *Proto*. For each server, $s \in S$, our goal is to uncover as many *distinct amplification-inducing query patterns* as possible (say AF $\geq$ δ=10) while keeping our network footprint low. These per-server patterns output by AmpMap can inform our use cases, such as assessing risk and informing defenses. Intuitively, each pattern is a template for describing protocol queries. In a given pattern, each field takes (1) a value or (2) a contiguous range. Queries in the same pattern trigger similar protocol behavior, and hence, have similar AFs (formal definitions in our extended technical report [**?**]).

We obtain the list of open servers implementing a given protocol from public services (Shodan [20], Censys [34]). We prune out inactive protocol servers or servers owned by the military or government. Each protocol is defined by a set of `fields` ($F = \{f_1 \cdots f_n\}$), and a set of `accepted values` for each field ($AV(f_1) \cdots AV(f_n)$). We obtain the protocol format from protocol specifications (e.g., RFCs). For instance, DNS defines fields such as DNSSEC, id, and their accepted values (e.g., DNSSEC takes a value from $\{0,1\}$). A valid query of *Proto* is a list of values for each field (f_i=$v_i \in AV(f_i)$) and returns a response. To avoid malformed queries that may impact server operation, we only consider valid queries. We do not include derived fields (e.g., checksum, count-related fields). Some fields take a value from a set of strings (e.g., domain for DNS, OID for SNMP). For these, we sample values. For DNS domain fields, we take popular domains and with different features (DNSSEC-enabled vs. not). To this end, we keep the set of values for these fields small (a few tens). For the fields that take a list of values (e.g., OID list for SNMP), we also specify a length of a list as an input (§4).

To keep our footprint and impact on servers low, we impose a total query budget for each server, B_{total} (400–1500, §5). We also consider additional precautions such as limiting the rate per server and avoiding malformed requests (§6.1).

Scope: We focus on *stateless* and *unicast* protocols (e.g., UDP) and stateless amplification strategies. Thus, stateful protocols (e.g., TCP-based [30,49]) and broadcast or multicast protocols (e.g., [50]) are out of scope. Additionally, stateful attack strategies that seed entries to a server and subsequently launch a high AF query are outside our scope; e.g., we do not consider an attacker who registers his own domain for DNS with many records to amplify the attack.

Challenges: We now discuss three key challenges in achieving our goal. To illustrate these concretely, we consider a

Fields: $F = \{f_1, f_2, f_3, f_4, f_5\}$
Accepted values for each field: $AV(f_i)$
1. f_1 takes a value from 0 to 1; $AV(f_1) = [0,1]$
2. f_2 takes a value from 0 to 99; $AV(f_2) = [0,99]$
3. f_3 takes a value from 0 to 65535; $AV(f_3) = [0,65535]$
4. f_4 takes a value from 0 to 7; $AV(f_4) = [0,7]$
5. f_5 takes a value from 0 to 1; $AV(f_5) = [0,1]$

Figure 4: Simplified protocol definition to highlight challenges of uncovering amplification queries

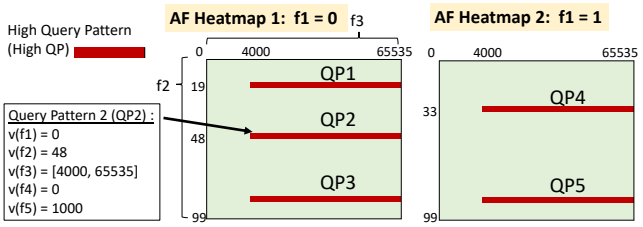


Figure 5: Query space for one server, s_1. QP_i refers to a query pattern

simplified protocol inspired by the structural properties of real protocols. The protocol is shown in Figure 4 and consists of 5 fields with their accepted values. Figure 5 represents the structure of amplification-inducing query patterns for a single server s_1 varying two of these fields, f_2 and f_3, while fixing the other three fields' values. The left side is when f_1=0, and the right side is when f_1= 1. In both cases, f_4 and f_5 are 0 and 1000, respectively. Each such "red" (darker) region in these heatmaps is a potential *query pattern*. Even this relatively simplified protocol highlights several key challenges. We observe these challenges across protocols we surveyed (especially for more complex protocols like DNS and NTP):

- **C1:** We observe a *large query space* of $2 \times 100 \times 65K \times 8 \times 2$ >200M values; i.e., it is infeasible to explore this space exhaustively.
- **C2:** Even for a single server, *the structure of amplification can be complex* as the fields in a query are dependent on each other and need to be simultaneously set. For instance, both f_2 and f_3 in QP_2 (Figure 5) need to be set to 48 and [4K, 65535], respectively, to yield high AF. Intuitively, in real protocols, such behavior occurs as certain flags need to be set to trigger a relevant behavior. For certain servers to yield large AF for DNS (§2.2), we need to set EDNS to 0 and rd to 1. Also, note the relationship between the query and AF does not necessarily have a nice continuous

structure. Worse, our goal is to uncover as many patterns as possible in this complex, multi-field search space, making the problem even more challenging.

- **C3:** Servers have *a large degree of variability.* As we saw in §2.2, the exact AF for a given query may differ, and the set of query patterns also may differ. Figure 6 shows the structure for three servers (including s_1) for the case when f_1 is 1. In our simplified protocol, queries in QP1 for s_1 incurs high AF for s_2 (i.e., QP1) but not for s_3. Due to the server configuration and the view of data a server has (e.g., the number of peers for the NTP server), s_3 does not have any query patterns that cause high AF.

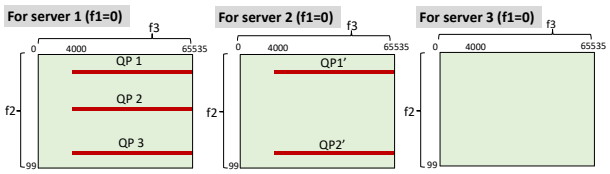


Figure 6: Query space across multiple servers, only showing the case when f_1=0. (The left-most heatmap for s_1 is the heatmap 1 in Figure 5.)

4 AmpMap Overview and Design

In this section, we discuss our key insights regarding the structural properties of amplification common to many protocols that enable our practical design. We start with a single server case (§4.1) and use that to build a multi-server solution (§4.2).

4.1 Single-Server Algorithm

Before we explain our insights, let us consider two seemingly natural baselines and see why these are not practical. (We empirically confirm this in §5.)

1. *Random fuzzing:* We can randomly pick a field value to construct a query. Unfortunately, achieving coverage across distinct patterns would be prohibitively expensive. For instance, if there are 10 patterns and the density of each pattern to the total query space is 0.1 (ε), we need at least 29K queries to discover all patterns. We present analysis in §A.
2. *Heuristic optimization techniques:* Existing heuristic optimization techniques (e.g., `Simulated Annealing`) may find only a few patterns. These are ill-suited to achieve coverage as these getting stuck in local optima.

4.1.1 Single-Server Insights

Next, we present our insights to make the problem tractable. At a high level, these insights were derived from a combination of simple analysis, local server experiments, and the measurements we saw in §2.2.

Insight 1 (I1): ***Amplification-inducing query patterns exhibit locality and overlap in their field values.***

Intuitively, we observe that query patterns often share a subset of specific field values. This structural property suggests that given a query, q, in one of the amplification-inducing query patterns, we may not need to change all N fields at a time. Instead, we can discover other nearby patterns by sweeping one field at a time. Conceptually, we can view the query space as a `logical graph` and look for "neighboring" queries that differ in the value of just one field to discover other patterns. Figure 7 shows a logical graph representation of the query space for the abstract protocol (Figure 5). In this graph, each node is a query and an edge between two queries, q, and q', indicates that they differ in only one field value (e.g., f_2). For instance, from a query in QP1, a simple per-field search approach, as described above, can discover queries in QP2 and QP3 by changing f_2. To discover QP5, we need to search f_1 from a query in QP3.

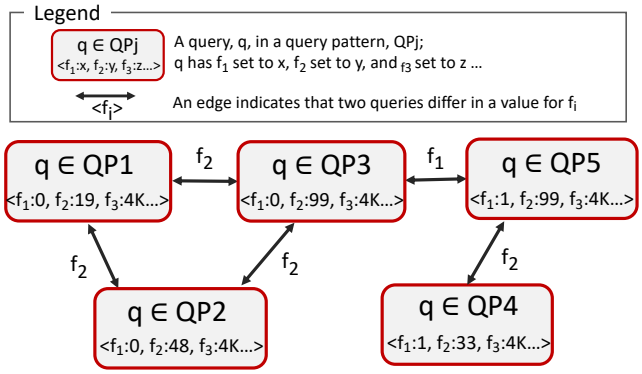


Figure 7: Viewing the query space as a logical graph (for the abstract protocol shown in Figure 5)

Insight 2 (I2): ***If the density of amplification-inducing queries is $>$ ε, then random sampling will likely find one such query using $\geq \frac{1}{\varepsilon}$ queries.***

This is a very simple probabilistic analysis insight. If the overall density of the queries that give high AF is ε, then the probability of picking one such query is ε. Then, the expected budget to find one such query is $\frac{1}{\varepsilon}$. For instance, if a probability of a picking an amplification-inducing query is $\frac{1}{1000}$, then we need an expected budget of 1000 samples. This analysis suggests a viable path to find at least one query in one of the amplification-inducing query patterns, which can subsequently be used to exploit the above locality structure.

Insight 3 (I3): ***Fields with large accepted value ranges either do not affect amplification or exhibit contiguous range structure w.r.t.*** AF.

Even if we use **I1** and only need to vary one field value at a time, we still may require a high query budget as some fields take a very large set of accepted values. Fortunately, many of the large-range fields tend not to affect amplification. If they

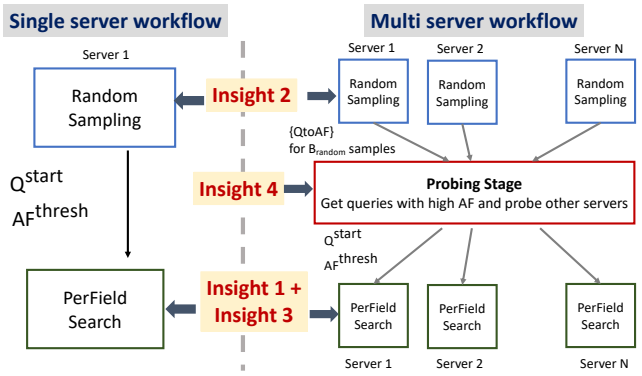


Figure 8: AmpMap Workflow

do, we observe that there is a large contiguous range (e.g., f_3 with [4K, 65535]) that exhibit similar behavior. For instance, as long as the EDNS payload is set to a large value (i.e., 4096), an EDNS feature will allow large responses. Thus, instead of exhaustive sweeping, we can sample values for large fields. Specifically, we use a logarithmically-spaced sampling strategy to get at least one query from a contiguous range if the ranges are sufficiently large.

Algorithm 1: AmpMap algorithm for a single server

Input: B: query budget, $AV(f_i)$ for $i = 1,..,n$: accepted value for each packet header field
Output: $QtoAF$: maps each query to corresponding AF

```
/* Step 1: Random Search                                   */
1 QtoAF = RunRandomUpdateMap(B_rand)
2 Q^start = FindTopKQueries(QtoAF, K = 1)
3 AF^thresh = ComputeThresh(QtoAF) /* Step 2: Local
    Search                                                 */
4 LocalSearchUpdateMap(QtoAF, Q^start, AF^thresh)
```

4.1.2 Single-Server Workflow

Putting the above insights together, we present our workflow for a single server (left side of Figure 8 and pseudo code in Algo. 1). Recall that we want to maximize coverage of distinct query patterns given a fixed query budget, B_{total}. Note that in choosing a value for B_{total}, we want to strike a balance between coverage and network load. Our goal is not to find optimal parameters, but to use reasonable ranges that work well in practice. We empirically find that 1200-1500 is a good operating range for relatively complex protocols like DNS, as we see diminishing returns beyond this (Figure 18 in §5.7). For simple protocols (with a smaller search space), this property still holds.

RandomSample Stage: Given a fixed B_{total}, the algorithm randomly samples B_{rand} queries to discover an amplification-inducing query (**I2**). The discovered queries are the starting points to run the next phase, per-field search, to improve coverage. For choosing a B_{rand}, we empirically observe that choosing 10% to 45% of the total budget is sufficient (Figure19a in §5.7). Recall that to leverage the locality (**I1**), we just need to find one (or a handful) query that induces amplification. As we will later, we use multi-server experiments to make this further robust to potential misestimation of the B_{rand} needed for a server, i.e., even when the RandomSample Stage fails to find a feasible starting point (§4.2.2).

Per-field search: We then run the Per-field search (Algo. 2) leveraging **I1**. It takes an input of $QtoAF$, which contains each query to the AF from the RandomSample Stage. We also need to determine other relevant input parameters.

- *Starting queries for the per-field search (Q^{start})*: We pick top K queries w.r.t. the AF values. Given the locality structure, we find choosing one starting query is sufficient.
- *The threshold to prune low* AF *queries (AF^{thresh})*: If neighboring queries have AF below AF^{thresh}, the per-field search prunes them from further exploration. If the value is too low, the search will degenerate into an exhaustive search. If too high, the search terminates without exploration. As a practical trade-off, if the maximum AF is above 2δ, we make the threshold to be δ (i.e., 10). If it is below 2δ, we use a threshold equal to some fraction of the maximum AF observed in the random stage (e.g., half).

Using each query from Q^{start}, the per-field search searches the neighboring queries by varying one field value (SearchNeighbor(...) referenced in Line 7; defined in Line 13 of Algo. 2). It uses a log-sampling for large fields and exhaustive search for other fields. Further, for fields that take a set of strings as an input (e.g., domains for DNS), we recommend inputting an accepted set as a small set (i.e., few tens). This is a conscious decision as such fields tend not to have a "contiguous" structure w.r.t. the AF, and each concrete value has a distinct semantic. Hence, we need to treat these fields as small fields (where we do an exhaustive search). For fields that take a list as an input (e.g., SNMP takes a list consisting of object identifiers or OIDs), we search over both the item (OID) and the size of the list. For this field type, it is worthwhile to see how the AF changes when this list size is large. Hence, we recommend putting a non-small value (i.e., ≥ 256) to log sample the values.

Avoiding already-visited patterns: We have one more practical challenge as each query pattern consists of tens of thousands of queries. Some field take ranges (e.g., f_3=[4000, 65535] in a pattern). If we naively explore, we may redundantly explore other queries in the same query pattern, wasting our query budget. To avoid this, we heuristically detect if we have already explored a pattern to decide if we can skip exploring this further. To do so, we infer the contiguous range of a field that incur above-the-threshold AF as we sweep each field (InferRange(...), defined in Line 24 of Algo. 2). When we need to explore a query, q', we first check whether q' has already been visited (IsNewPattern($\cdots$), referenced in Line 5) and only explore if it was not. We refine the inferred pattern structure during the per-field search as

Algorithm 2: Per-Field Search and Helper Functions

```
1  Function PerFieldSearch(QtoAF, Q^start, AF^thresh):
2      Q^explore = {Q^start}; PatternsFound = {}
3      while Q^explore is not empty do
4          q ← Extract from Q^explore
5          if IsNewPattern(q.pattern , PatternsFound) then
               /* Search neighbors for a new pattern   */
6              PatternsFound.insert(q.pattern)
7              tmpQtoAF = SearchNeighbor(q, AF^thresh)
8              QtoAF.insert(QtoAF^neighbor)
9              Q^explore = Q^explore ∪ tmpQtoAF.keys()
10         else
               /* if not new, skip exploration        */
11             MergeQueries(q.pattern, PattersFound)
12     return QtoAF
13 Function SearchNeighbor(q, AF^thresh):
14     NeighborQToAF = {}
15     foreach protocol field f_i do
16         Q_i = {q[f_i ← v_i], for v_i ∈ Values_i}
17         QtoAF_i = SendQuery(q ∈ Q_i)
           /* Merge queries into contiguous ranges with
              high AF                                  */
18         HighRanges = InferRange(q, Values_i, QtoAF_i,
            AF^thresh)
           /* Find representative sample from each range
              */
19         for ⟨v_l, v_r⟩ ∈ HighRanges do
20             patternid = q.pattern[f_i ← (v_l, v_r)]
21             q_n = q[f_i ← rand([v_i, v_r])]
22             NeighborQToAF.append( q_n → AF_n )
23     return NeighborQToAF
24 Function InferRange(q, Values_i, QtoAF, AF^thresh):
25     IsCurRangeActive = False; HighRanges = { }
26     CurStart = CurEnd = NULL
27     for v ∈ Values_i sorted in ascending order do
28         if IsCurRangeActive then
29             if QToAF_j ≥ AFThresh then
30                 CurEnd = v
31             else
32                 IsCurRangeActive = False
33                 HighRanges.append(⟨ CurStart,CurEnd ⟩ )
34         else
               /* we encounter a new high range        */
35             if QtoAF_j ≥ AFThresh then
36                 IsActive = True; CurStart = CurEnd = v
       /* If still active, include the last entry      */
37     if IsCurRangeActive then
38         HighRanges.append( ⟨ CurStart, v ⟩)
```

we get a new range that contains the old range. The search terminates if the budget is exhausted or there are no more queries to explore.

Let us look at a concrete example using the abstract protocol presented in §3. Suppose we are currently exploring a query q, ⟨ f1:0, f2:48, f3:6000 $\cdots$⟩, from a QP 2. When it is a turn to explore f_3, we *log sample* f_3 to obtain the AFs and find that [5K, 65535] has contiguously "high" AFs. Then, we use this range to describe the pattern (i.e., ⟨ f1:0, f2:48, f3:[5K, 65535] $\cdots$⟩). We first check whether this is contained in already-visited patterns and only explore if not already visited. We present the analysis for a single server in §A.

4.2 Multi-Server Algorithm

We now discuss how we extend the insights and workflow from a single-server case to handle the multi-server case.

4.2.1 Multi-Server Insights

Insight 4 (I4): ***While servers exhibit variability, some share a subset of amplification-inducing queries.***

Recall the abstract protocol on multiple servers in Figure 6. In that example, the queries in QP1 for s_1 also incur high amplification for s_2 but not for s_3. While these servers are not identical in all query patterns that induce amplification, some of these servers can share a subset of query patterns (even if the specific AF values may differ). We also have observed this in our small-scale experiment in §2. Specifically, while the similarity of query patterns between a pair of servers is low, it is not always 0. This is natural as these servers implement the same protocol. This property allows us to further reduce overhead by sharing insights across servers. That is, we can use already-found amplification-inducing queries (from the RandomSample Stage) and probe other servers using these queries. This probing increases the probability of having a good starting point to run the per-field search for each server. Note that our workflow still accounts for server heterogeneity (while sharing insights across servers) as we still run the per-field search for each server.

4.2.2 Multi-Server Workflow

We start with the RandomSample Stage per server as in the single-server case. The key addition is a new stage called the `Probing Stage` (Figure 8), which ensures that the insights are shared across servers. Specifically, using the high-amplification queries found for each server from the RandomSample Stage, we test them on other servers to increase the chance of finding good starting queries for each server.

Probing Stage: Turning this idea into practice, we take all queries that give high AFs across servers from the RandomSample Stage. Then, we pick a small number of queries to probe other servers (say B_{probe} queries). A relevant question is how many queries to use for B_{probe}. We observe that anywhere between 5% to 30% of the total budget is sufficient, where we chose 10% (validation in §5.7). We do not want to assign too much for this value to ensure a sufficient available budget

Algorithm 3: AmpMap algorithm for multiple servers

Input: B_{total}: query budget
$AV(f_i)$ for $i = 1,..,n$: accepted value for each packet field
S: a set of servers
Output: PerServerQToAF : maps each query to corresponding AF

```
PerServerQToAF = {} /* Step 1: Random Search */
for s ∈ ServerSet do
    RunRandomUpdateMap(B_rand, PerServerQToAF[s])
/* Step 2: Pick probes based on current obs. */
Q^probe = PickProbes( PerServerQtoAF , B_probe )
/* Run additional probes per server */
for s ∈ S do
    ProbeQToAF_s = SendQuery(Q^probe)
    PerServerQToAF[s].insert(ProbeQToAF_s)
/* Step 3: Per-field search for each server */
for s ∈ S do
    Q_s^start = FindTopKQueries(PerServerQToAF[s],K)
    AF^thresh = ComputeThresh(PerServerQToAF[s])
    PerFieldSearch(PerServerQToAF[s], Q_s^start, AF^thresh)
return PerServerQToAF
```

for other (critical) stages. Specifically, the Probing Stage is designed to supplement the RandomSample Stage for specific servers where the RandomSample Stage was could not discover amplification-inducing queries. The next relevant question is *how* to pick these probing queries. Consider a strategy where we pick the top-X queries w.r.t. the AF. This strategy may "overfit" to a specific query pattern or certain servers with many AF-inducing queries. We want to use a *diverse set of probing queries*. To this end, we take all queries with AF above the threshold, δ, and then run a simple K-means clustering where we conservatively set the number of clusters, K (e.g., 20).[5] To achieve diversity of patterns, we sample queries such that we have at least one query from each cluster, and for the remaining ones, we uniformly sample queries proportional to the cluster size. Here, the key for boosting the coverage is the fact that we use probing queries (Figure 19b in §5.7); the number of clusters is less critical.

The rest of the algorithm mirrors the single-server approach to pick starting points and run the per-field search. However, the input parameters (i.e., Q^{start}, AF^{thresh}) are server-specific to account for server diversity. The only difference is that the top-K starting points are based on the original set of random queries and the new additional B_{probe} queries. Note that for fields that take a set of strings (e.g., domain for DNS), we do not split the query budget across different field values (e.g., different domains). However, given that the per-field search does not favor queries with higher AF (as long as AF $\geq AF^{thresh}$), our algorithm does not bias one particular field value (e.g., a particular domain) over another. Further, as we will see in §5.3, we combine the queries across all servers to infer patterns. Combining data allows us to infer patterns despite having a small per-server budget (e.g., 1500).

[5]To run K-means clustering, we define our custom distance function. We normalize N fields and then bin the large fields

5 Evaluation

In this section, we present findings from our Internet measurements for 6 UDP-based protocols (DNS, NTP, SNMP, Memcached, Chargen, SSDP) and local testing for 3 protocols (QOTD, Quake, RPCbind). In contrast to a scoped experiment in §2.2, the results here cover more protocols, servers and search over the packet header space (opposed to sending a fixed set of queries). We also validate our design against strawman solutions and parameter choices.

	# IPs Scanned (a)	# Pruned IPs (b)		# IPs Taken (c) = (a)+(b)	# IPs in DB (d)	% IPs Scanned (c) / (d)
		Invalid Proto	Gov't Mil.			
DNS	10K	18,698	15	28,713	8.02M	0.36
NTP ^{OR}	10K	4317	5	14,322	8.4M	0.17
NTP ^{AND}	3,083	234,374	7	237,464	8.4M	0.28
SNMP ^{OR}	10k	4,933	3	14,936	2.16M	0.69
SNMP ^{AND}	10K	60,187	9	70,196	2.16M	0.33
Memchd	10K	11,736	9	21,745	63K	3.5
Chargen	10K	68,065	6	78,071	83K	9.4
SSDP	10K	78,617	3	88,620	2.16M	3.3

Table 4: Statistics on (a) the # of IPs we scanned per protocol, (b) the # of pruned IPs, (c) the # of raw IPs we needed from the DB ; (d) the # of total public-facing IPs as is (Shodan and Censys); and (e) the % of IPs we scanned

Measurement setup: We use nodes from CloudLab [33], where 1 node is used as a controller, and 30 as measurers.[6] For these 6 protocols, we scanned 10K *sampled* servers for each protocol: DNS with OPT records for EDNS, NTP, SNMP, Memcached, Chargen, SSDP. For DNS, we scan the servers obtained from Censys and, hence, these are mostly open resolvers.[7] As the protocol formats for SNMP's *Get*, *GetNext*, and *GetBulk* requests differ, we treated each as a separate protocol and ran separately. Similarly, we ran separate runs for NTP's mode 7 (private), mode 6 (control), and mode 0-5 (normal). We obtained public server IPs from Censys [34] and Shodan [20]. We randomly sampled IPs from these lists and pruned out inactive servers (e.g., those that do not respond to *dig* for DNS) or owned by the military or government. For certain protocols (SNMP, NTP) that have different modes of operation with distinct formats, we consider two notions of active server, whether the server responds to (1) "any" of the modes (OR filter); or (2) "all" of them (AND filter). We present results for both schemes, using AND/OR superscripts to denote each (e.g., SNMP ^{AND}).

To finish our measurements in a few days and restrict the number of (shared) nodes we use, we target 10K servers per protocol.[8] Table 4 shows: (1) the number of IPs we needed from Shodan and Censys to get our final server lists,[9] (2) the

[6]We restricted our node usage to 31 per experiment, as CloudLab is a shared platform across institutions

[7]We can easily extend AmpMap to handle authoritative servers.

[8]We could not obtain 10K servers for NTP ^{AND}.

[9]For DNS, we posit that many are inactive because the Censys DB was from Jan 2020 when the measurements were conducted in May 2020.

total number of public-facing IPs for each protocol (as of May 30, 2020) from Censys (for DNS) and Shodan (for others); and (3) the % of IPs we scanned from the Internet. When we refer to servers to present our results, we are referring to sampled servers rather than the entire Internet servers.

In our experiments, each server is pinned to a measurer. We do not spoof IP addresses, and we send legitimate queries and listen to responses. We impose a limit of 1 query per 5 s for each server with a timeout of 2 seconds (i.e., 7 seconds per query). This rate gives approximately 3 days to complete for 10K servers as 30 measurers can handle 500 servers at a given time.[10] Our network load is low: 48 kbps (egress) across all measurers and 1.6 kbps per measurer. If we assume an average AF of 5, then we incur 240 kbps in ingress bandwidth.

Protocol specifics: For protocols with more than 10 fields (DNS, NTP, RPCbind), we used a query budget of 1500 queries per server, setting 45% for RandomSample Stage and 10% for the Probing Stage. For simpler protocols, we used a budget of 400 queries with the same budget split. For QOTD, Quake, RPCbind, we set up a single CloudLab server running the protocol. Some fields, such as domain fields for DNS, took strings. As discussed in §4.1.2, we picked 10 popular domains[11] spanning different industry sectors, and enabled features (e.g., DNSSEC supported vs. not). For SNMP, we pick v2's OIDs based on the RFC up to depth 4 (i.e., A.B.C.D). For fields that take as input a list of values (e.g., an OID for SNMP), we also search over the list's length.

5.1 Protocol and server diversity

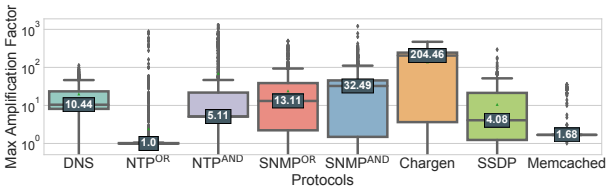
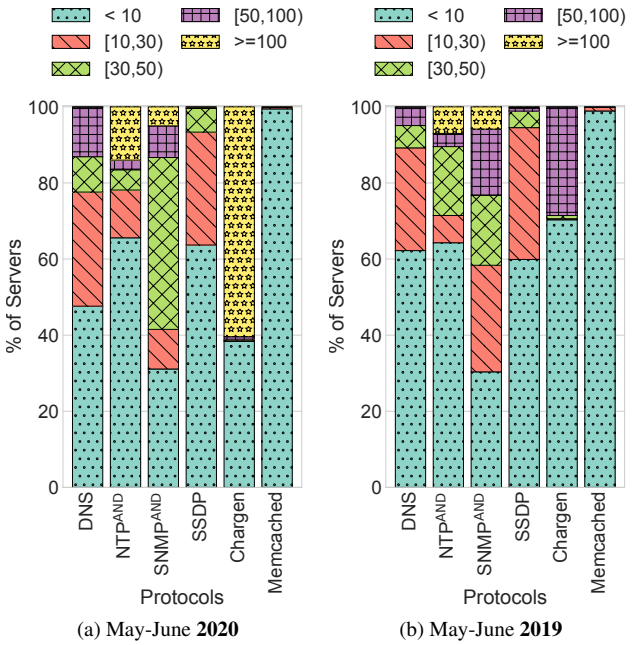


Figure 9: Boxplot showing the distribution of the maximum AF achieved by each server given a protocol

Finding 1: ***There is significant variability in the maximum amplification a server can yield across servers.***

Figure 9, where y-axis is log-scale, shows the distribution of the maximum AF achieved by each server for each protocol. (For SNMP and NTP, we combine the results across different modes.) For many protocols, we observe a long tail in the distribution. For instance, while the median for SNMP ^{OR} is 13.01 AF, the maximum is 495. While the median is 1 AF for NTP ^{OR}, the maximum is 860. For NTP ^{AND}, while the median is 5.11 AF, the maximum is as large as 1300! This high variability confirms we cannot simply count the number of open servers or attribute the same risk to each server.



(a) May-June **2020** (b) May-June **2019**

Figure 10: Summary across servers and protocols (from 2019 and 2020 runs)

Finding 2: ***There is substantial variability in the maximum*** AF ***distribution across protocols.***

Figure 10a shows the maximum AF distributions with varying AF ranges (e.g., 10-30) across protocols; these experiments ran in May–June 2020. For SNMP and NTP, we only show the results for AND schemes for brevity. First, protocols vary in the percentage of potential amplifiers with AF $\geq$ 10: 52% for DNS, 34% for NTP ^{AND}, 69% for SNMP ^{AND} ... 0.6% for Memcached. Further, protocols differ in the most common AF ranges ($\geq$ 10) that servers can yield. AF range for DNS is concentrated on 10 to 30 but above 100 for Chargen. For NTP ^{AND}, 14% of servers give above 100 AF. These results suggest that measuring the risk should take into account the AF distribution per protocol.

Finding 3: ***There is variability across time in the*** AF ***distribution across servers for different protocols.***

Figure 10b shows the maximum AF distribution from measurements done in `2019`, as opposed to `2020` for Figure 10a. (Across two runs, there are minor differences in the AmpMap parameters such as 53% budget for the RandomSample Stage in 2019 vs. 45% in 2020, but they do not really affect the results.) These figures visually highlight the differences across the two years. For instance, only 7% of NTP ^{AND} servers yielded AF $\geq$ 100 in 2019 vs. 14% in 2020. 90th percentile of DNS servers induced above 30 AF in 2019 but above 59 AF

[10] Each run takes 3 hours (7s$\times$1500 queries) and need 69 hours to handle 10K servers (not accounting for timeouts).

[11] berkeley.edu, energy.gov, chase.com, aetna.com, google.com, Nairaland.com, Alibaba.com, Cambridge.org, Alarabiya.net, Bnamericas.com

in 2020 (almost doubled) using the identical domain lists. We acknowledge that as we sample servers, we cannot attribute the root cause of differences, i.e., the change in server list vs. the actual attack landscape. However, such variability is the reason that calls for the need to do continuous (periodic) measurements rather than a one-time analysis.

5.2 Assessing amplification risks

	Known Pattern	Risk Quantification		Results
		Prior Work	AmpMap	
DNS	EDNS:0,ANY [1,57]	287K	149K	1.9× ↑
	EDNS:0,ANY,TXT [57,62]	Unknown	183K	N/A
DNS (domains w/o DNSSEC)	ANY,TXT [57,62]	Unknown	126K	N/A
NTP OR	monlist [2,57]	5,569K	13K	427× ↑
NTP AND	monlist [2,57]	5,569K	635K [12]	8.8× ↑
SNMP OR	*GetBulk* [3,57]	64K	223K	3.5× ↓
SNMP AND	*GetBulk* [3,57]	64K	317K	5× ↓
Chargen	Request	3588K	1399K	2.9× ↑
SSDP	Search [3,57]	308K	126K	2.7× ↑
Memcached	Stats [3,17]	100M [3]	18K	5.6K × ↑

Table 5: Contrasting the risk extrapolated from prior works and measured by AmpMap for 10K servers

Finding 4: ***Even for known patterns, extrapolations (e.g., [32, 57]) mis-estimate amplification risk.***

Table 5 summarizes the known patterns and their corresponding risks assessed using AmpMap and prior works [1, 57] (same risk used in §2.2). For AmpMap, given a pattern for each protocol (e.g., `monlist` for NTP), we calculate the total risk across 10K servers using the Eq. 1. We find that the baseline techniques from prior work have *significant mis-estimation*. For instance, these techniques overestimate NTP by 427×, underestimate SNMP v2 by 3.5×, and overestimate Chargen by 2.9×. The large inaccuracy of 427× overestimation for NTP is because the previously reported AF of 556 [57] does not generalize to most NTP servers. Our findings confirm a study of NTP amplification [32], which specifically focuses on the `monlist` feature. Further, the underestimation of 3.5× for SNMP is because the prior analysis (by assuming a fixed query) does not account for *polymorphic variants*. Specifically, we can achieve higher amplification using *GetBulk* requests with varying OID fields and the number of OIDs to request. While the previously reported average of the worst 10% servers for *GetBulk* requests (SNMP) is 11.3 AF [57], the average of the worst 10% from our measurement dataset is 90 for SNMP OR (7.9× larger than 11.3), and 97 AF for SNMP AND (8.6× larger).

Finding 5: ***Prior recommendations (e.g., [32, 57]) miss many query patterns and leave substantial residual risk.***

We now quantify the risks from *new patterns* that will be missed by prior analysis (Table 6). For DNS, there are other combinations of EDNS and record type fields that yield large

	New Patterns	Risk Quantification
DNS	¬(EDNS:0 ∧ ANY lookup)	3274K (21.9 × known pattern)
	¬(EDNS:0∧(ANY ∨TXT) lookup)	3127K (17.1 × known pattern)
NTP OR	req code ≠ monlist (20,42)	43K (3.3 × known pattern)
NTP AND	req code ≠ monlist (20,42)	663K (1 × known pattern)
SNMP OR	*GetNext*	61K (0.27 × known pattern)
	Get	10K (0.04 × known pattern)
SNMP AND	*GetNext*	101K (0.32 × known pattern)
	Get	11K (0.03 × known pattern)
SSDP	None	0
Memcached	*Get*, *Gets*	33K (1.9 × known pattern)

Table 6: Amplification risk from new patterns whose risks will be missed by prior analysis

(and considerable) amplification. The total risk from these other patterns (e.g., record types: `LOC`, `URI` lookups) across 10K servers is 3,274K. This unforseen risk is 21.9× larger than the risk of known patterns (149K)! Figure 11 shows a bird's-eye view of the residual risk. We observe similar trends for other protocols. For instance, for NTP, a collective risk from other features (e.g., `get restrict`) is 276× higher risk than the known risk. For simpler protocols like SSDP, our measurements do not reveal new patterns.

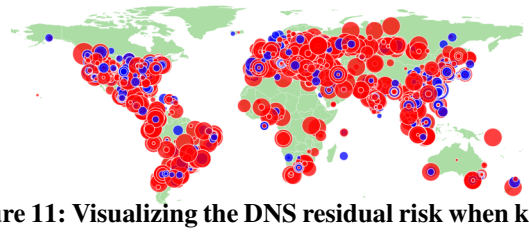


Figure 11: Visualizing the DNS residual risk when known patterns (EDNS**:0 and** record type**:**ANY |TXT**) are blocked. The size of the circle ∝ the max** AF **of each server. Red circles denote when the delta is ≥ 20%.**

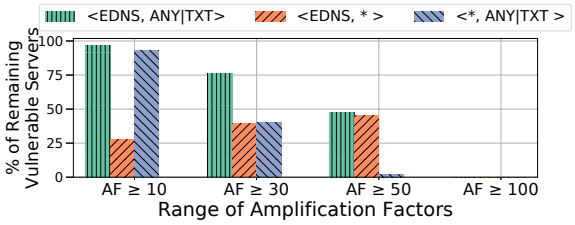


Figure 12: % of DNS servers that remain susceptible to amplification even if we use recommendations by prior works to block query patterns; i.e., ⟨ EDNS**,** ANY |TXT ⟩ **is a filter that blocks queries EDNS:0 and** ANY **or** TXT **lookups.**

Next, we conduct *what-if* analysis to analyze what percentage of servers are susceptible to amplification if we were to block known patterns. Given that prior works do not provide concrete signatures, we consider a few possible interpretations, i.e., a combination of EDNS:0 and record type:ANY or TXT. Figure 12 shows that even with EDNS:0 and (ANY or TXT) lookups blocked, more than 97% of servers still can yield AF greater than 10. For NTP (mode 7), even with `monlist` as a signature,[13] 30.5% servers can still yield AF ≥ 10 and 4.8%

[13] A follow-up paper mentioned the possibility of other settings that induce

$\geq$ 100! We observe similar trends for SNMP. However, prior recommendations achieve high coverage for SSDP, Chargen, and Memcached.

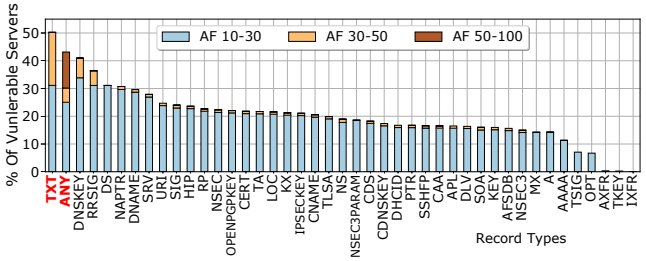


Figure 13: The variability of field values (for a specific field, record type**) that contribute to high amplification. Apart from known ones** (record type:`ANY`, `TXT`)**, many other** record type **values can lead to large** AF**.**

5.3 In-depth analysis on DNS

The previous discussion suggests there are many patterns not highlighted by prior work. We analyze this further, focusing on DNS here and deferring other protocols to §5.4-§5.6.

We start with a record type field as this field determines `ANY` vs. `NS` record lookups. Figure 13 shows the percentage of servers that can induce considerable AF for each possible value of this field. While the top-2 record types are `TXT` and `ANY` (pointed by prior work), more than 20% of our sampled servers can yield more than 10 AF with 19 other record type values (e.g., `URI`, `HIP`, `RP`, `LOC`, `CNAME`). Some of these (e.g., `NAPTR`) incur very high AF, especially if used in conjunction with the DNSSEC (DNSSEC-OK) set. While many DNSSEC-related record type values (e.g., `RRSIG`, `DNSKEY`) can yield high AF [61], we also observe many record type values "unrelated" to DNSSEC (e.g., `NAPTR`, `SRV`). This finding is significant — even if we block `ANY`, `TXT` queries, there are many other types that can induce high amplification.

Summarizing and analyzing query patterns: The above analysis only considers one field. In practice, many other combinations of fields are susceptible, and we want to understand the structure of amplification-inducing query patterns (QPs). For this summarization, we considered several standard data mining techniques (i.e., hierarchical clustering, K-means clustering, decision trees) but found that none were suitable.[14]

Given this, we designed a custom heuristic (Figure 14). Starting from AF-inducing queries across all servers, we generate a set of candidate patterns where some fields are set to concrete values or ranges, and others are wildcarded. Specifically, for large fields (e.g., id, payload for DNS) we identify candidate *ranges* by dividing the accepted values for a large field into exponentially-spaced bins (e.g., $\{[0,10], [11,100]$ $...\}$. Then, for *each server*, we generate a bit vector (e.g., 1111)

amplification, they did not specify which request types [32].

[14]Clustering assume that we know the number of clusters or the right distance metric/threshold. Given the large combinatorial space, decision trees produce uninterpretable outputs.

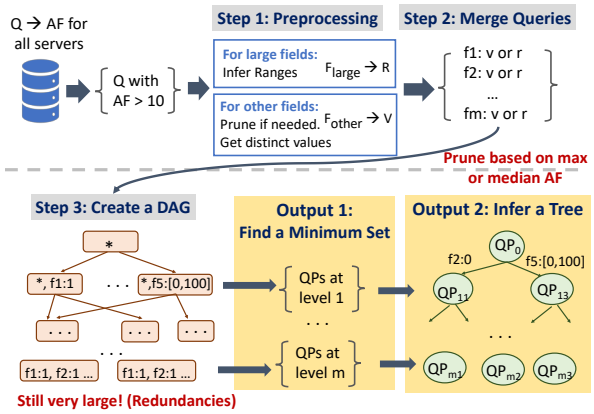


Figure 14: Steps to obtain query patterns to shed light on the patterns of amplification

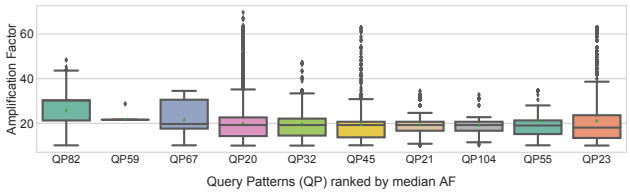
to represent these bins; a bit is set to 1 if a server has a query with AF $\geq$10 using a field value that belongs to the bin range. Finally, given a set of bit vectors for all servers, we take candidate vectors that are observed across at least 10% of servers. We prune out fields that appear not to affect amplification; i.e., we count the number of queries (with AF $\geq$ 10) by checking if wildcarding the field makes the AF value histogram follow a uniform distribution. We then generate candidate patterns by generating all combinations of values and ranges. From these candidates, we prune out QPs with AF less than 10 based on the *maximum* or the *median* AF. We represent the QPs as a logical `Directed Acyclic Graph` (DAG), with these QPs are leaf nodes (Step 3, Figure 14). We create a parent node by taking one of the nodes in the current *level* and wildcarding one field; the DAG root is a node where all fields are wildcards. Given this DAG, we consider two analysis:

1. *Minimum set cover per level (Output 1, Figure 14)*: We compute the minimum set-cover of QPs at each level that logically covers all leaf nodes; e.g., the set of QPs obtained at level 10 represents the *minimum set of* QPs to describe QPs using only 10 fields as concrete values or ranges.
2. *Hierarchical analysis (Output 2, Figure 14)*: To see dependencies across fields, we create *a tree* where the edge is annotated with the field and its value, which became concrete as we increase the level (an example in Figure 16).

We run the above procedure separately for (1) domains with DNSSEC support, and (2) domains without support.

Corollary 1: ***Many unexpected patterns lead to high*** AF***, e.g., with*** DNSSEC ***off and unrelated to*** `ANY` ***records.***

DNSSEC-related patterns: Figure 15a shows a boxplot of the top-10 QPs w.r.t. the median AF when 8 fields are left concrete (level 8). QP 82 incurs the largest median AF of 30 with $\langle$ EDNS:0, payload:*, record type:`RRSIG`, rd:* $\cdots\rangle$. In this pattern, it is *not* necessary to have a rd set to 1 and shows that `RRSIG` lookups can also cause high AF. The rank-2 QP has EDNS set to 1 and not 0, which is a known pattern. In



(a) Rank based on median AF

ID	Field Values
QP82	⟨ EDNS:0, payload:*, record type:RRSIG,ad:1,rd:*,rcode:8 ⋯ ⟩
QP20	⟨ EDNS:1, payload:*, record type:*, ad:0, rd:1, rcode:* ⋯ ⟩
QP32	⟨ EDNS:1, payload:*, record type:TXT, ad:0, rd:1, rcode:* ⋯ ⟩

(b) Describing query patterns (QPs)

Figure 15: DNS: Top 10 query patterns for a particular depth where 8 fields are left as concrete values of ranges

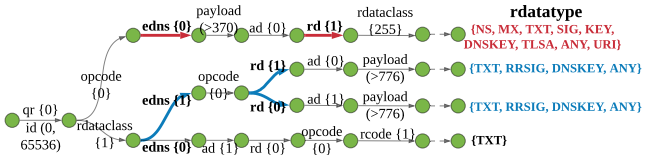


Figure 16: Tree showing how the query patterns change across levels. An edge means a field value transitioned from a wildcard (*) in level L to a concrete value or range in the next level, $L+1$.

fact, several servers that yield high AF had EDNS not set to 0. Further, as we find many record type values that lead to high AF (also seen in Figure 13), this QP has a record type set to *. Further, as a side note, when we were pruning out fields that appear not to affect AF (Figure 14), a DNSSEC-OK field got pruned out. However, we observed that setting this bit to 1 on certain queries can induce high AF on some servers.

Non-DNSSEC patterns: For certain servers, domains without DNSSEC support can yield high AFs. The median AF for the top-1 QP is 21 with ⟨ EDNS:1, record type:TXT, rd:1 ⋯⟩. This confirms that TXT records can cause high AF [62]. We also observe record type values such as DS appear among the QPs; some are attributed to anomalous servers.

Corollary 2: ***There are many query patterns that, while not maximum, provide high enough amplification. Hence, focusing on only one or a handful of patterns can render existing mitigation (i.e., [41]) ineffective.***

At each level of the DAG, more QPs are concentrated at AF between 10 and 20. At the leaf nodes, 699 query patterns produce a median AF of 10 to 20 while only 47 above 20 AF. Purely focusing on one pattern or a handful to drive the mitigation plan will be insufficient.

Corollary 3: ***There are complex dependencies across field values inducing high*** AF ***change based on other fields.***

The DAG output (Figure 14) shows complex dependencies across field values that yield high AF. Specifically, Figure 16 shows a subset of a tree (for DNSSEC-related) where the QPs are filtered based on the "median" AF. If we consider a top branch with EDNS:0 and rd:1, with NS, MX, ⋯ TLSA, URI record types cause high AF. Some other combinations (i.e., blue edges) will cause different record type values to induce high AF. Surprisingly, we find a non-trivial number of servers that yield high AF even when rd (recursion desired) is 0 (off)! These suggest that (1) there are many combinations of *multiple* fields values that lead to high AF, and (2) this finding generalizes to many servers (as QPs are kept if the median AF across servers is $\geq$ 10 AF). Further, if we consider a tree where QPs are pruned based on the maximum AF (less aggressive pruning), we see even more combinations leading to high AF (e.g., OPENPGPKEY, SOA record types).

Further, we observe that not all servers behave according to specifications, further adding to variability in QPs. For instance, when EDNS:0 is used, the response should be chopped to the specified EDNS payload value. Unfortunately, for many servers, this is not the case. For instance, 88 servers out of 10K yield AF above 50 with payload less than 512. During our 2019 measurements, we saw 311 AF for one server (for SRV records lookup), where we saw many IP fragments. This server went offline shortly after the experiment. While DNS over UDP does use IP fragmentation to deliver large payloads [15], this makes defenses more difficult as they miss key fields such as port information [4].

Vendor	# of Total Servers	# of Server (AF $\geq$ 10)	% of Servers (AF $\geq$ 10)
Bind	946	236	24.9%
Dnsmasq	917	819	89.3%
Version:recursive-main/*	522	12	2.3%
Microsoft	261	250	95.8%
PowerDNS	78	50	64.1%
unbound	40	26	65%

Table 7: Statistics on the affected DNS vendors

Corollary 4: ***Given the variability of query patterns, blocking the top-K percentage of patterns still leave significant residual risk; i.e., the 50th percentile of servers has 80% or more residual risk, even with blocking 20% of query patterns (infeasible in practice).***

We now analyze the percentage (%) of the residual risk if we had used the top-K percentage (%) of QPs to block these queries. For this analysis, from the inferred QPs (Figure 14), we do not prune them based on the maximum or median AF; we need to know all QPs that lead to high AF for each server. We take the top-5 and 20% of these 11K QPs (sorted by their median AF) and use them to block amplification-inducing queries from each server. Unfortunately, we observe that even blocking the top-20% QPs (which is infeasible in practice) still leaves 50% of the servers with an 80% risk or higher

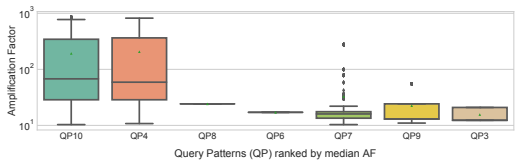


Figure 17: NTP top query patterns, where the top-2 are `monlist` patterns. Other top QPs have peer list, if reload, peer list sum, and peer stats as req code.

(blocking the top-5 % leaves 96.7% risk or higher).

Corollary 5: ***Many DNS vendors are affected.***

Table 7 shows the affected vendors with servers that can yield AF $\geq$ 10. We only show vendors with more than 20 servers. We discuss our efforts to notify these vendors of the vulnerability in § 6.2.

5.4 Amplification patterns for NTP

We discuss amplification patterns for NTP. As we do not discover new patterns for mode 0-6,[15] we focus on mode-7 (private mode). Recall that we need to prune candidates QPs based on maximum or median AF (Figure 14). As we observe a high variance across AF achieved by different NTP servers, we looked at the QPs where they are pruned based on the maximum. Figure 17 shows these QPs (pruned based the maximum AF) where they ranked by the median AF. Apart from `monlist` (QPs 10 and 4), we observe request codes of `peer list`, `if reload`, `peer list sum`, and `peer stats` from NTP $^{\text{OR}}$. Some of these other QPs can yield as large as a few hundred as seen by the long tails in Figure 17. From NTP $^{\text{AND}}$ servers, we also observe `mem stats`, `if stats`, and `get restrict`. Our findings again complement **Corollary 2**. Furthermore, the software versions (with servers that can yield $\geq$ 10AF) are 4.1.1-2, 4,2,4, 4,2,6-8, and 4.2.0. We observe that the servers that can induce high AF with other request codes (other than `monlist`) are not particularly tied to one single version but span across multiple versions.

5.5 Amplification patterns for SNMP

We now discuss SNMP patterns, which have 3 modes of operations, i.e., *GetBulk*, *GetNext*, and *Get*. We start with ***GetBulk***, which is a known pattern [3] (reported average of 6.3 AF [57]). However, our measurements revealed *polymorphic variants* that lead to significantly higher AFs. For instance, we saw an average of 22.4 AF for SNMP $^{\text{OR}}$ and 31.8 AF for SNMP $^{\text{AND}}$, which are higher than the reported. Specifically, an attacker can modify OID value and the number of OIDs to yield higher AFs. We generally observe higher AF for query patterns with (1) a single-digit OID (near the root) such as 2, 1, 0, and (2) a list containing multiple OID (i.e., 2-15 but above 15). However, given server variability, there are exceptions. For example, an OID of 1.3.6.1.2, and a list size of 1 appears in one of the top-4 patterns. The top-1 QP from the SNMP servers yields a median AF of 35 with ⟨ community:public ··· OID:2, numoid: (0,8) [16]⟩. From SNMP $^{\text{AND}}$ servers, the top-1 QP yields 45 median AF with OID:0.

Vendors	# Total Servers	GetBulk		GetNext	
		# Server (AF≥10)	% Servers (AF≥10)	# Server (AF≥10)	% Servers (AF≥10)
net-snmp	5357	5044	94.2%	3445	64.3%
cisco Systems	594	96	16.2%	60	10.1%
Sonic Wall	220	21.7	98.6%	27	12.3%
Broadcom Corp.	205	193	94.1%	81	39.5%

Table 8: Statistics on the affected SNMP vendors

We now discuss ***GetNext*** requests. While only *GetBulk* has been highlighted in the prior analysis, AmpMap discovers that a single *GetNext* request can also yield hundreds of AF (similarly, by varying the OID and the number of OIDs). From SNMP $^{\text{AND}}$ servers, 37% of servers can yield AF above 10 and 0.74% above 100 AF! From SNMP $^{\text{OR}}$ servers, 10% servers yield above 10 AF and 0.14% above 100 AF. However, unlike SNMPbulk, we saw high AFs for various OIDs (e.g., 1.3.6.1.2, 0, 1); this is expected because *GetNext* just requests the next variable in the tree, unlike a *GetBulk* request, which requests several *GetNext* requests. Note that while we also replicated that a local server can yield 15 AF with *GetNext* by varying the list size, we posit that we see higher AF in the wild given server variability. Table 8 shows the affected vendors for servers using *GetBulk* or *GetNext* requests. We only show for vendors with more than 200 servers, combining the results from both SNMP $^{\text{AND}}$ and SNMP $^{\text{OR}}$ servers. Similar to DNS and NTP, this amplification vulnerability affects multiple vendors and not just one.

Lastly, measurements reveal that ***Get*** requests also can yield tens of AF (but not as large as *GetNext*). From SNMP $^{\text{OR}}$, 0.73% servers that have AF greater than 10. Unlike *GetNext* patterns, we observe high AF for OID of 1.3, and 1.3.6.1.3-4.

5.6 Amplification patterns for other protocols

SSDP: Amplification risk is inherent with SSDP's "discovery" feature. Our inferred QPs are quite simple. For QPs pruned based on the median AF, we see a discovery request with one UUID of `ssdp:all`. This is expected as this feature fetches "all" UUID information. However, for QPs based on the maximum AF, we see many UUIDs leading to $\geq$ 10 AF. Again, this confirms the presence of multiple query patterns.

Memcached: We did not find any QPs that lead to above 10 AF other than the "stats" request (a known pattern) from our 2020 run. If we use our runs from 2019, some of the QPs with *get* and *gets* requests did induce above 10 AF. However, it

[15] There was one packet that incurred high AF for mode-6 but this packet contained many ICMP redirects so we do not report this.

[16] More accurate version is (2, 8) but our range inference is a heuristic.

is still the case that "stats" are by far the dominant pattern, and the residual risk from *get* and *gets* requests are negligible. Further, while the known AF for Memcached is tens of thousands [24], the maximum we find from our 2020 run is 35 AF (we believe many have been patched or taken offline).

Chargen: As Chargen servers respond to any UDP datagram, the QPs learned at the leaf nodes contain all possible characters and lengths. We represented the search space as a list of hex strings where we search over the hex character and the length of the hex character.

We validate the existence of amplification-inducing query patterns for three protocols in a lab setting. For these, we confirm the known patterns but do not find additional ones.

Quake: "Get status" message induces AF of 10 in our setting.

QOTD: As this server responds with random quotes, we see higher AF with smaller list sizes and larger quote size.

RPCbind: The request for the process number running on the server with a correct version ID incurs high AF (i.e., 10).

5.7 Parameters and Validation

Given the lack of ground-truth for all servers, we use a combination of local-server experiments, a large-scale simulation, and example measurements for validation. In the local experiment, we randomly sampled 2M queries on a local DNS server and measured the AFs to infer the signatures (§5.3). Our simulator models an amplification function that maps a query to AF based on (1) field types, (2) the # of servers, (3) the # of pattern structures across servers, (4) the # of pattern for each (3). For (3), indicating 100 pattern types instantiates 100 graph structures across servers where each gets mapped to one type. (3) simulates the pattern variability across servers.

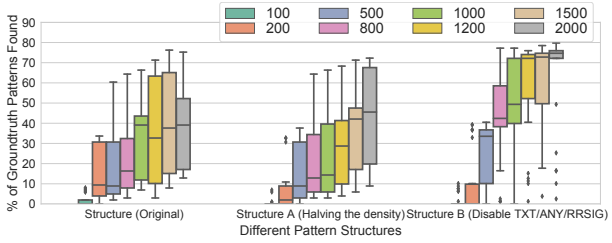
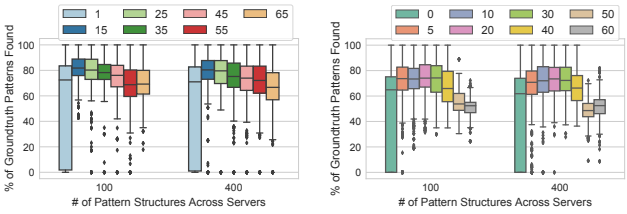


Figure 18: Validating the choice of total budget (B_{total})

Validating parameters: There are three key parameters: (1) per-server total budget, B_{total}, (2) allotting B_{total} across different stages (e.g., Probing Stage), and (3) the number of clusters for K-means.

To see the impact of the total budget (B_{total}), we use the local DNS server experiment. Fixing other parameters (50% for B_{rand}), we varied the B from 100 to 2000 (Figure 18). To show the robustness across multiple pattern structures, we "emulated" different pattern structures given one setup. We emulated the effect of (A) reducing the % of AF-inducing queries by half (emulating this by adding "dummy" field



(a) % of budget for B_{rand} (b) % of budget for B_{probe}

Figure 19: Validating the choice of budget allocation

entries that yield 0 AF), and (B) disabling certain patterns (`TXT`, `RRSIG`, `ANY` lookups). Clearly, using only a few hundred achieves low coverage but starts seeing the diminishing return at 1200 or 1500. We chose 1500 for complex protocols (e.g., DNS). This experiment shows that our chosen B_{total} is in a sufficiently good operating region.

To see the impact of the budget across stages, we use our simulator with 1K servers. We configured 30% of servers not to induce high amplification (similar to the real-world). To analyze the robustness w.r.t. different levels of diversity, we test against 100 to 400 pattern structures. First, using 50% for B_{rand}, we vary the B_{probe} from 0 to 40% (Figure 19b). Using 0% for probing hurts coverage but using 5% and 30% is robust across settings. We chose 10% (lower end of the range) to spare the budget for other (more critical) stages. Similarly, we vary the B_{rand} from 0 to 70% (Figure 19a). We observe robustness across 5% to 45%. As it is crucial for this RandomSample Stage to discover at least one AF-inducing query (for most servers), we chose 45% (the higher end). This leaves a per-field search with the remaining 45%.

To validate the number of clusters, we use the same simulator and evaluate based on the % of servers, which the chosen B_{probe} discovered at least one high AF query. Then, we vary the number of clusters from 2 to 200 and observe robustness across these values; i.e., this is not a crucial factor.

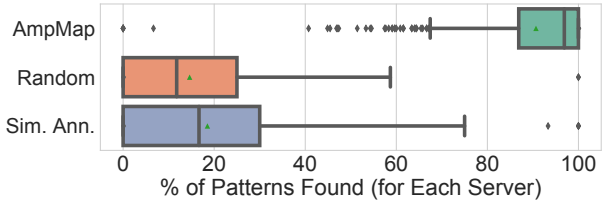


Figure 20: Validation of coverage of AmpMap and alternate solutions using 1K server measurements

Comparing alternatives: We compare AmpMap vs. two baselines: 1) Simulated Annealing (SA), and 2) pure random search. Our success metric is pattern coverage across a set of servers. We compared these solutions using small-scale 1K measurements. As we lack the ground-truth for each server, we compare the relative performance across these solutions rather than to claim optimality or completeness. Using a query

budget of 1500, we inferred the signatures combining the output across all solutions. Then, we analyze the coverage for each server. For a given server, we take all queries with AF $\geq$ 10 across three solutions, which serves as the basis of comparison for this server. Then, for each strategy, we compute the % of patterns discovered for each server. Figure 20 shows the coverage across 1K servers. While SA performs better than pure random strategy, the median coverage is 16.7%, while the pure random strategy has an 11.9% median. AmpMap achieves 97% coverage in this relative comparison.

6 Precautions and Disclosure

We carefully considered the impact of our measurements and the disclosure of our findings. We followed the ethical principles (Menlo Report [27]) and the scanning guidelines suggested by prior efforts (Zmap [35]). At a high-level, we adhered to these principles of (1) *minimizing the harm* by taking multiple measurement precautions (§6.1), and (2) *being transparent in our method and results* by informing relevant stakeholders of our findings and explaining the purpose of our scanning (e.g., when we send out email notifications) (§6.2).

6.1 Scanning precautions

We took precautions to ensure that there was no harm to the servers and the network. Our study was approved by IRB under non-human subject criteria. We took care to ensure that our measurements do not burden servers or the Internet.

- We send at most one query per 5 seconds, do not send malformed requests, and cap overall budget per server.
- We do not scan the IPv4 network space but only known public servers obtained from Censys [34] and Shodan [20].
- We do not spoof the source IPs to induce responses to others. Our measurers explicitly receive the responses.

Abuse complaints: We worked closely with the CloudLab [33] administrators whom we notified of our measurements and the purpose of AmpMap. We only received one abuse complaint from running back-to-back SNMP small-scale experiments (500 servers) on June 3, 2020. This complaint came from a third-party monitoring framework called `greynoise.io` [12]; their goal is to notify the probing activities in the Internet and mass scanners (e.g., Censys [34], Shodan [20]) are also likely to be flagged by them [12]. We resolved this abuse complaint by discussing this with Cloudlab admins. We did not receive any other abuse complaints from our 10K server measurements. Across all 6 protocols, we also ran small-scale runs (300 servers) from our public-facing server. We are not aware that the campus network operators received any abuse complaints from these measurements.

6.2 Disclosure

Next, we discuss our steps for responsible disclosure to relevant stakeholders.

Protocol		# Sent	# Resp	Protocol		# Sent	# Resp
DNS		4335	49	SNMP AND	bulk	4433	36
NTP OR	priv	112	0		next	2387	34
	normal	2	0		get	26	2
NTP AND	priv	915	4	SSDP		3563	6
SNMP OR	bulk	4007	30	Chargen		6008	9
	next	1670	11	Memcached		51	0

Table 9: Statistics on the # of notification emails we sent and the responses we got from system owners

SUBJECT: Vulnerable DDoS Amplifier

BODY: Security researchers at Carnegie Mellon University have been conducting Internet measurements to quantify the risk of amplification distributed denial-of-service (DDoS) attacks. Our team has noticed your system, IP with $PORT$ running $PROTOCOL$, can be abused to create an amplification attack (US-CERT). That means certain network queries can induce large responses (i.e., amplification factor as defined by US-CERT). Note that this may or may not be a result of mis-configuration of the server. An example of a network packet that can cause an amplification factor greater than 10 is: $PACKET INFO$.
Please feel free to contact us at ampmap.proj@gmail.com should you have any questions and/or concerns. The details and motivation of our project can be found in $OUR WEB$.

Figure 21: A sample notification email to IP owners

Notifying IP owners: We notified the IP owners whose servers can induce AF greater than 10. Following best practices, we obtained the abuse and/or contact email from WHOIS [51]. We include an example notification sent from a project's email, ampmap.proj@gmail.com in Figure 21. Table 9 shows the number of emails we sent and human (not automated) responses we got; e.g., for DNS, we send 4335 emails and received 49 responses. Example responses include "Thanks $\cdots$ service detected on *ADDR* has been shutdown the time to install necessary mitigation" and "We were not even aware this was the case. we have disabled SNMP." We also received detailed responses such as "The server is operated by one of our downstream sites ... this server gives an upward referral instead of returning SERVFAIL or REFUSED. This is consistent with particular implementation of DNS server (and IMO, it's wrong, for exactly the reasons you state ...)"

Vulnerability reporting: We have initiated a process of disclosing our findings to the affected parties mediated by the CERT® Coordination Center (CERT/CC). CERT/CC has accepted our coordination request and is in the process of identifying and notifying the affected parties. Our findings require multi-party coordination because unexpected amplification is potentially a protocol issue, and so all relevant vendors need to be notified in a consistent manner. Further, we have tested the effectiveness of the Response Rate Limiting (RRL) [41], a mitigation feature for DNS amplification attacks. We informed the vendor that having multiple patterns can partially degrade the performance (more details in §8).

Notifying the vendors: Our vulnerability reports to CERT/CC specify affected vendors for DNS, SNMP, and NTP. CERT/CC is initiating the conversation with the ven-

dors so that we can share the packet captures and commands that elicit large amplification.

7 Related Work

Amplification attack and mitigation: Many network protocols have amplification vulnerabilities [54]. Rossow [57] discovered amplification vulnerabilities in 14 UDP-based protocols by manually analyzing the code and the binary. Follow up research also analyzed detailed amplification vector in specific protocols by focusing on a specific set of features (e.g., analyzing DNSSEC in DNS [61], `monlist` in NTP [32]). However, using AmpMap, we found many other record type values that can incur high AF. Some have looked at TCP-based amplification [30, 49], which is outside the scope of AmpMap. There is also an active discussion on the mitigation of amplification attacks (e.g., [6, 8, 14]). Jonker et al., [44] have done a measurement study on the adoption of these DDoS protection services [44]. Further, some orthogonal efforts focus on monitoring (e.g., [43, 47]) and linking DDoS services (e.g., [48]). For instance, prior work [43] leverages data from multiple Internet infrastructures (e.g., backscatter traffic, honeypots) to macroscopically characterize DDoS attacks (including amplification attacks), attack targets, and mitigation behaviors. Our work is inspired by these prior efforts. Specifically, our goal is not in characterizing attacks or linking attacks that are happening in the wild. Instead, to the best of our knowledge, AmpMap is the first to study the problem of *automatically* mapping Internet-wide amplification vulnerabilities by precisely identifying query patterns that can induce large amplification.

Protocol implementation testing and verification: There is a rich literature on testing and verification of protocol implementations. Bishop et al. [29] develop a practical specification-based testing technique for both TCP and UDP based network protocols; PIC [55] applies symbolic execution for checking interoperability in protocol implementations; Kothari et al. [46] apply symbolic execution for manipulation attacks. Recent work [45, 53] also applied model checking techniques for protocol implementations. Our work is different from this line of work because of our specific focus on uncovering amplification vectors rather than protocol bugs.

Existing machine learning techniques: The problem that AmpMap tackles can be also viewed as a black-box optimization problem. Hence, one interesting future work is to leverage and customize these techniques for AmpMap's purpose, e.g., derivative-free optimization [36,42,56] or Bayesian Optimization that can optimize for a black-box function. For instance, we would need to customize these algorithms to achieve coverage rather than finding the maximum value and also handle server diversity. These efforts can benefit from our observations and insights. Further, the current AmpMap algorithm can also benefit from parameter tuning, e.g., automatically decide the % spent on the RandomSample Stage based on the density observed so far.

Fuzz testing: Our technique is closely related to a large body of work on fuzz testing of software [52]; some well-known tools are DART [38], SAGE [40], grammar-based fuzzing [39], mutational fuzzing [65], among many others (see [59]). Some have been applied for testing protocol implementations; e.g., [25, 28] focus on finding security flaws in the SIP protocol, and [60] focuses on security protocols. However, these approaches focus more on safety bugs (e.g., memory). While our technique is a form of fuzzing, we tackle a different application domain that will benefit from a different set of domain-specific insights.

Message format extraction: AmpMap currently assumes that the protocol formats are known. As such, our work can benefit from prior work on message format extraction and protocol model inference (e.g., [31, 63])

8 Countermeasures

In this section, we discuss countermeasures against amplification DDoS attacks in light of our findings in §5. More extensive countermeasures are discussed by Rossow [57] and we omit them for brevity.

Response rate limiting: As a response to UDP-based amplification attacks, an authoritative name servers should, and mostly do, use response rate limiting (RRL) [1]. The idea of RRL is to limit the number of requests that a server sends to a client, so the server cannot be used to reflect an attack on the client [57]. Popular DNS servers already support this feature [41]. In light of our findings that revealed multiple query patterns, we revisit the effectiveness of the RRL mitigation. Given that the implementation of RRL focuses on identical response and client identity, it calls into a question of RRL's effectiveness if an attacker rotates multiple patterns. To test, we set up a local DNS authoritative bind server (9.16) and obtained amplification-inducing queries using AmpMap. Then, we varied (1) the number of distinct queries to rotate (37 vs. 2111), and (2) the inter-query time (0 vs. 0.05 s). We compared the total response bytes (within a window of 15 s) and the average AF when the RRL feature is on vs. off. Our results reveal that using multiple query patterns and carefully controlling the inter-query time can degrade the performance of RRL and give an adversary power. Specifically, if an attacker uses more patterns (2,111 instead of 37) and an appropriate inter-query time (0.05 s), the average AF even when the RRL is on is 92% that of the case when RRL is off. However, by using a larger inter-query time, an attacker consequently generates less attack traffic. That is, an adversary will need to trade off between the efficiency of an attack vs. the total bandwidth of the attack. Understanding this trade-off is an exciting research direction to explore. In light of our findings, what we need is more advanced RRL going forward. Given the diversity of patterns, it is unclear whether focusing on the exact query or exact response is the right mechanism.

Secure configuration and setups: Network operators and device vendors can help mitigate some of these threats by either taking the server offline (for legacy protocols) or changing configurations. For instance, certain network devices (e.g., network-enabled printers) have SNMP on by default, and fixing these configurations could help mitigate these threats. Our experiences in informing IP owners show that multiple cases when operators were unaware that their devices are publicly accessible. Furthermore, the suggested best practice for public-facing DNS servers is to restrict access to only authorized clients. While we also advocate following the best practices, mitigating these attacks is unfortunately not as simple. Even in the perfect scenarios where all the servers are correctly configured, our measurements uncovered valid features within a protocol exploitable for attacks. Therefore, a long-term solution is to carefully consider the protocol design choices or design protocols that are correct-by-construct.

9 Conclusions

Given the constant evolution of protocols, server implementations, we need a systematic approach to map the DDoS amplification threat. This paper bridges this gap by synthesizing structural insights with careful measurement design to realize a low-overhead service called AmpMap. AmpMap can systematically confirm prior observations and also uncover new-possibly-hidden amplification patterns that are ripe for abuse. As future work, we plan to add support for more protocols and expand the scale of measurement to make this a continuous "health monitoring" service for the Internet.

Acknowledgements

We thank the anonymous reviewers, Nicolas Christin, and Min Suk Kang for their helpful suggestions. We thank the artifact evaluation committee for their efforts and suggestions, and Devdeep Ray and Ankit Jena for their help with an earlier version of AmpMap. This work was also supported in part by: NSF awards CNS-1440065 and CNS-1552481; the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the U.S. Army Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] Alert (TA13-088A) UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA13-088A.

[2] Alert (TA14-013A) NTP Amplification Attacks Using CVE-2013-5211. https://www.us-cert.gov/ncas/alerts/TA14-013A.

[3] Alert (TA14-017A) UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA14-017A.

[4] Broken packets: IP fragmentation is flawed. https://blog.cloudflare.com/ip-fragmentation-is-broken/.

[5] *CyberGreen*. https://stats.cybergreen.net/.

[6] Ddos and security resource center. https://tinyurl.com/y8h7o9vw.

[7] *DDoS Attacks Get Bigger, Smarter and More Diverse*. https://tinyurl.com/ydzdnfur.

[8] Dns reflection defense. https://tinyurl.com/lbffebt.

[9] *DNS SURVEY: OPEN RESOLVERS*. http://dns.measurement-factory.com/surveys/openresolvers.html.

[10] Executive Order 13800 - Strengthening the Cybersecurity of Federal Networks and Critical Infrastructure. https://www.govinfo.gov/content/pkg/DCPD-201700327/pdf/DCPD-201700327.pdf.

[11] *Flooding the web: The internet's epic attack amplification problem*. https://tinyurl.com/ycjnqg9n.

[12] Grey Noise. https://greynoise.io/about.

[13] *Here's how much money a business should expect to lose if they're hit with a DDoS attack*. https://tinyurl.com/y7s45ls3.

[14] How to defend against amplification attacks. https://tinyurl.com/yb5gotte.

[15] IPv6, Large UDP Packets and the DNS. http://www.potaroo.net/ispcol/2017-08/xtn-hdrs.html.

[16] Memcrashed - Major amplification attacks from UDP port 11211. https://tinyurl.com/yatp4649.

[17] Open Memcached Key-Value Store Scanning Project. https://memcachedscan.shadowserver.org/.

[18] Security Bulletin: Crafted DNS Text Attack. https://tinyurl.com/y9zpevuy.

[19] ShadowServer. https://www.shadowserver.org/.

[20] SHODAN. https://www.shodan.io/.

[21] Technical Details Behind a 400Gbps NTP Amplification DDoS Attack. https://tinyurl.com/mcf32xg.

[22] The DDoS That Almost Broke the Internet. https://tinyurl.com/pl26tw3.

[23] The Spoofer Project. http://spoofer.cmand.org.

[24] UDP-Based Amplification Attacks. https://www.us-cert.gov/ncas/alerts/TA14-017A.

[25] H. J. Abdelnur, R. State, and O. Festor. Kif: A stateful sip fuzzer. In *Proc. IPTComm*, 2007.

[26] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable internet protocol (aip). In *Proc. ACM SIGCOMM*, 2008.

[27] M. Bailey, D. Dittrich, E. Kenneally, and D. Maughan. The menlo report. *IEEE Security Privacy*, 10(2):71–75, 2012.

[28] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna. Snooze: Toward a stateful network protocol fuzzer. In *Proc. ISC*, 2006.

[29] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets. In *Proc. ACM SIGCOMM*, 2005.

[30] K. Bock, A. Alaraj, Y. Fax, K. Hurley, E. Wustrow, and D. Levin. Co-opting Firewalls for TCP Reflected Amplification. In *Proc. USENIX Security*, 2021.

[31] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proc. USENIX Security*, 2007.

[32] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir. Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proc. IMC*, 2014.

[33] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proc. ATC*, 2019.

[34] Z. Durumeric, D. Adrian, A. Mirian, M. Bailey, and J. A. Halderman. A search engine backed by internet-wide scanning. In *Proc. CCS*, 2015.

[35] Z. Durumeric, E. Wustrow, and J. A. Halderman. Zmap: Fast internet-wide scanning and its security applications. In *Proc. USENIX Security*, 2013.

[36] D. E. Finkel. Direct optimization algorithm user guide. 2003.

[37] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Appl. Math.*, 39(3):207–229, Nov. 1992.

[38] P. Godefroid et al. Dart: Directed automated random testing. In *Proc. PLDI*, 2005.

[39] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[40] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[41] Internet Systems Consortium. Using the response rate limiting feature. https://kb.isc.org/docs/aa-00994, 9 2018.

[42] K. G. Jamieson et al. Query complexity of derivative-free optimization. In *Proc NIPS*, pages 2672–2680, 2012.

[43] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, and A. Dainotti. Millions of Targets Under Attack: a Macroscopic Characterization of the DoS Ecosystem. In *Proc. IMC*, 2017.

[44] M. Jonker, A. Sperotto, R. van Rijswijk-Deij, R. Sadre, and A. Pras. Measuring the adoption of ddos protection services. In *Proc. IMC*, 2016.

[45] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proc. NSDI*, 2007.

[46] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding protocol manipulation attacks. In *Proc. ACM SIGCOMM*, 2011.

[47] L. Krämer, J. Krupp, D. Makita, T. Nishizoe, T. Koide, K. Yoshioka, and C. Rossow. Amppot: Monitoring and defending against amplification ddos attacks. In *Proc. RAID*, 2015.

[48] J. Krupp, M. Karami, C. Rossow, D. McCoy, and M. Backes. Linking amplification ddos attacks to booter services. In *Proc. RAID*, 2017.

[49] M. Kührer, T. Hupperich, C. Rossow, and T. Holz. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In *Proc. USENIX Security*, 2014.

[50] S. Kumar. Smurf-based distributed denial of service (ddos) attack amplification in internet. In *Proc. ICIMP*, 2007.

[51] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *Proc. USENIX Security*, 2016.

[52] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990.

[53] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proc. NSDI*, 2004.

[54] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *SIGCOMM CCR*, 31(3):38–47, July 2001.

[55] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing Protocol Implementations for Interoperability. In *Proc. NSDI*, 2015.

[56] L. M. Rios and N. V. Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[57] C. Rossow. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *Proc. NDSS*, 2014.

[58] T. Rozekrans and J. de Koning. Defending against DNS reflection amplification attacks. https://tinyurl.com/bvw3d85.

[59] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[60] P. Tsankov, M. T. Dashti, and D. Basin. Secfuzz: Fuzz-testing security protocols. In *2012 7th International Workshop on Automation of Software Test (AST)*, 2012.

[61] R. van Rijswijk-Deij, A. Sperotto, and A. Pras. Dnssec and its potential for ddos attacks: A comprehensive measurement study. In *Proc. IMC*, 2014.

[62] R. Vaughn and G. Evron. Dns amplification attacks preliminary release. 2006.

[63] Y. Wang, X. Yun, M. Z. Shafiq, L. Wang, A. X. Liu, Z. Zhang, D. Yao, Y. Zhang, and L. Guo. A semantics aware approach to automated reverse engineering unknown protocols. In *Proc. ICNP*, 2012.

[64] R. Weber. Better than Best Practices for DNS Amplification Attacks. https://tinyurl.com/y75u32ju.

[65] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling black-box mutational fuzzing. In *Proc. CCS*, 2013.

A Formal Analysis

We present analysis sketch of why AmpMap can discover medium-to-high modes and compare it with other strawman solutions. To make analysis easier, we make two simplifying assumptions: (1) We only consider a single-server case (§4.1.2); and (2) The ratio of the number of high AF queries to the total number of possible queries, d , is known.

Definitions: We first give necessary definitions for the formal analysis. We use query ranges to denote a set of queries. Particularly, we write a query range QR as $\langle f_1 : [v_1^l, v_1^r], f_2 : [v_2^l, v_2^r], \ldots, f_n : [v_n^l, v_n^r] \rangle$, where $v_i^l, v_i^r \in AV(f_i)$ and $v_i^l < v_i^r$ for $i = 1, ..., n$. A query range represents a set of queries in a natural way. A query $q = \langle f_1 = v_1, .., f_n = v_n \rangle$ is in QR (written $q \in QR$) iff $v_i^l \leq v_i \leq v_i^r$ for $i = 1, .., n$.

Given a constant δ, a δ-high query pattern (or simply high query pattern if δ is clear from the context) QP is a query range $\langle f_1 : [v_1^l, v_1^r], f_2 : [v_2^l, v_2^r], \ldots, f_n : [v_n^l, v_n^r] \rangle$ satisfying the following two conditions: 1) all queries in the query range induce high AF. That is, $\forall\, q \in \text{QP}, AF(q) \geq \delta$; 2) the specified range of each field in QP is a maximal in terms of inducing high AF. That is, $\forall\, i = 1, .., n$, $v_i'^l$ and $v_i'^r$, if $v_i'^l < v_i^l \leq v_i^r \leq v_i'^r$ or $v_i'^l \leq v_i^l \leq v_i^r < v_i'^r$, then $\exists$ a query $q \in \langle f_1 : [v_1^l, v_1^r], \ldots, f_i : [v_i'^l, v_i'^r], \ldots, f_n : [v_n^l, v_n^r] \rangle$ such that $AF(q) < \delta$.

Given a protocol, *Proto*, we assume that the set of all high query patterns of *Proto* is unique. We denote the set of all high query patterns as $\mathbb{P}_{Proto}$.

Given a *Proto* and a total budget, Q, the covered high query pattern by Q, denoted $co(Q)$, is the set of high query patterns of *Proto* where each high query pattern shares at least one query with Q. That is, $co(Q) = \{\text{QP} \in \mathbb{P}_{Proto} | Q \cap \text{QP} \neq \emptyset\}$. Based on this definition, we can now formally state the goal of AmpMap. Given a server s running protocol *Proto*, *AmpMap* seeks to maximize the size of $co(Q)$.

A.1 Analysis of strawman approaches

Here, we analyze the expected budget for different strategies for the one-server case.

Exhaustive Search: An exhaustive search enumerates valid queries of the protocol. While this can discover all patterns, the budget is prohibitively large: $E(B) = \prod_{i=1}^{N} |AV(f_i)|$, where N is a number of fields.

Random Search: For pure random search, the expected number of queries to cover all high query patterns is: $E(B) = \int_0^\infty (1 - \prod_{i=1}^{|\mathbb{P}|} (1 - e^{-p_i t}))dt$

Here, p_i is the probability of picking a query in the i-th high query pattern [37]. The expected budget increases exponentially as $|\mathbb{P}|$ increases.

A.2 Analysis of AmpMap approach

Under some simplifying assumptions we can analyze the expected budget to discover all patterns. To make analysis easier to present, we make three simplifying assumptions: (1)

Each field, f_i, is of homogeneous size F; (2) Each distinct pattern just has one query; and (3) We know the number of distinct patterns, NumPatterns.

In reality, our goal is to discover as many as possible. At a high-level, we can show that our worst-case run time is linear in the NumPatterns $\times F$. First, note that given d, the density of queries that give high AF, the expected budget to find one query in one of the patterns is $\frac{1}{d}$. Second, note that the number of queries required to sweep the all neighboring queries from a given query is $F \times NumField$.

Given these preliminaries and our assumptions on the "locality" structure, we can consider the best-case and worst-case analysis to discover all patterns. The best-case is when all patterns form a fully connected clique, where two queries in two distinct patterns are neighbors. This means, that when we start from a query in a q_1, we will discover all other NumPatterns-1 patterns in just one sweep. The worst case is when all 4 distinct patterns (QP1 $\cdots$ QP4) form a chain. That is, we need to do one sweep to discover an additional mode. Note that we are guaranteed to find another pattern (Observation **1**) because all patterns can be reached by sweeping each field. Hence, we need to do NumPatterns $-$ 1 sweep. Since we assume we know what is NumPatterns, our search will terminate when we discover all patterns. Taken together, the best-case run-time is $\frac{1}{d} + F \times NumField$, and the worst-case run-time is $\frac{1}{d} + (\text{NumPatterns} - 1) \times F \times NumField$.