UNIVERSITÄT
DES
SAARLANDES

Using Honeypots to Trace Back Amplification DDoS Attacks

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von

Johannes Krupp

Saarbrücken
2021

Tag des Kolloquiums:	21. Dezember 2021
Dekan:	Prof. Dr. Thomas Schuster

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Raimund Seidel
Berichterstattende:	Prof. Dr. Christian Rossow
	Prof. Dr. Andreas Zeller
	Prof. Dr. Katsunari Yoshioka
Akademischer Mitarbeiter:	Dr. Ahmad Ibrahim

Abstract

In today's interconnected world, Denial-of-Service attacks can cause great harm by simply rendering a target system or service inaccessible. Amongst the most powerful and widespread DoS attacks are amplification attacks, in which thousands of vulnerable servers are tricked into reflecting and amplifying attack traffic. However, as these attacks inherently rely on IP spoofing, the true attack source is hidden. Consequently, going after the offenders behind these attacks has so far been deemed impractical.

This thesis presents a line of work that enables practical attack traceback supported by honeypot reflectors. To this end, we investigate the tradeoffs between applicability, required a priori knowledge, and traceback granularity in three settings. First, we show how spoofed attack packets and non-spoofed scan packets can be linked using honeypot-induced fingerprints, which allows attributing attacks launched from the same infrastructures as scans. Second, we present a classifier-based approach to trace back attacks launched from booter services after collecting ground-truth data through self-attacks. Third, we propose to use BGP poisoning to locate the attacking network without prior knowledge and even when attack and scan infrastructures are disjoint. Finally, as all of our approaches rely on honeypot reflectors, we introduce an automated end-to-end pipeline to systematically find amplification vulnerabilities and synthesize corresponding honeypots.

Zusammenfassung

In der heutigen vernetzten Welt können Denial-of-Service-Angriffe große Schäden verursachen, einfach indem sie ihr Zielsystem unerreichbar machen. Zu den stärksten und verbreitetsten DoS-Angriffen zählen Amplification-Angriffe, bei denen tausende verwundbarer Server missbraucht werden, um Angriffsverkehr zu reflektieren und zu verstärken. Da solche Angriffe jedoch zwingend gefälschte IP-Absenderadressen nutzen, ist die wahre Angriffsquelle verdeckt. Damit gilt die Verfolgung der Täter bislang als unpraktikabel.

Diese Dissertation präsentiert eine Reihe von Arbeiten, die praktikable Angriffsrückverfolgung durch den Einsatz von Honeypots ermöglicht. Dazu untersuchen wir das Spannungsfeld zwischen Anwendbarkeit, benötigtem Vorwissen, und Rückverfolgungsgranularität in drei Szenarien. Zuerst zeigen wir, wie gefälschte Angriffs- und ungefälschte Scan-Datenpakete miteinander verknüpft werden können. Dies ermöglicht uns die Rückverfolgung von Angriffen, die ebenfalls von Scan-Infrastrukturen aus durchgeführt wurden. Zweitens präsentieren wir einen Klassifikator-basierten Ansatz um Angriffe durch Booter-Services mittels vorher durch Selbstangriffe gesammelter Daten zurückzuverfolgen. Drittens zeigen wir auf, wie BGP Poisoning genutzt werden kann, um ohne weiteres Vorwissen das angreifende Netzwerk zu ermitteln. Schließlich präsentieren wir einen automatisierten Prozess, um systematisch Schwachstellen zu finden und entsprechende Honeypots zu synthetisieren.

Background of this Dissertation

This dissertation is based on three peer-reviewed papers published at CSS 2016 [P1], RAID 2017 [P2], and IEEE EuroS&P 2021 [P3], with a fourth one currently under submission at USENIX Security 2022 [P4]. I contributed to all of them as the main author, and in two cases [P1, P3] as the sole contributor next to faculty-level co-authors.

Author's Papers for this Dissertation

[P1] **Krupp, J.**, Backes, M., and Rossow, C. Identifying the Scan and Attack Infrastructures Behind Amplification DDoS Attacks. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, 1426–1437. URL: https://doi.org/10.1145/2976749.2978293.

[P2] **Krupp, J.**, Karami, M., Rossow, C., McCoy, D., and Backes, M. Linking Amplification DDoS Attacks to Booter Services. In: *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*. Springer. 2017, 427–449. URL: https://doi.org/10.1007/978-3-319-66332-6_19.

[P3] **Krupp, J.** and Rossow, C. BGPeek-a-Boo: Active BGP-based Traceback for Amplification DDoS Attacks. In: *6th IEEE European Symposium on Security and Privacy*. 2021. URL: https://publications.cispa.saarland/3372/.

[P4] **Krupp, J.**, Grishchenko, I., and Rossow, C. AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities (and Using Those to Synthesize DDoS Honeypots). 2021. Under submission (USENIX Security 2022).

Further Contributions of the Author

[S1] Krämer, L., **Krupp, J.**, Makita, D., Nishizoe, T., Koide, T., Yoshioka, K., and Rossow, C. AmpPot: Monitoring and Defending Against Amplification DDoS Attacks. In: *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*. Springer. 2015, 615–636. URL: https://doi.org/10.1007/978-3-319-26362-5_28.

[S2] **Krupp, J.**, Schröder, D., Simkin, M., Fiore, D., Ateniese, G., and Nürnberger, S. Nearly Optimal Verifiable Data Streaming. In: *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*. 2016, 417–445. URL: https://doi.org/10.1007/978-3-662-49384-7_16.

[S3] Fleischhacker, N., **Krupp, J.**, Malavolta, G., Schneider, J., Schröder, D., and Simkin, M. Efficient Unlinkable Sanitizable Signatures from Signatures with Re-randomizable Keys. In: *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*. 2016, 301–330. URL: https://doi.org/10.1007/978-3-662-49384-7_12.

- [S4] Döttling, N., Fleischhacker, N., **Krupp**, **J.**, and Schröder, D. Two-Message, Oblivious Evaluation of Cryptographic Functionalities. In: *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*. 2016, 619–648. URL: https://doi.org/10.1007/978-3-662-53015-3_22.
- [S5] Jonker, M., King, A., **Krupp**, **J.**, Rossow, C., Sperotto, A., and Dainotti, A. Millions of targets under attack: a macroscopic characterization of the DoS ecosystem. In: *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*. 2017, 100–113. URL: https://doi.org/10.1145/3131365.3131383.
- [S6] **Krupp**, **J.** and Rossow, C. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, 1317–1333. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.
- [S7] Toorn, O. van der, **Krupp**, **J.**, Jonker, M., Rijswijk-Deij, R. van, Rossow, C., and Sperotto, A. ANYway: Measuring the Amplification DDoS Potential of Domains. In: *17th International Conference on Network and Service Management, CNSM 2021, Izmir, Turkey, October 25-29, 2021*. 2021.

Acknowledgments

This thesis would not have been possible without the help and support of many wonderful people who I would like to thank.

First, I want to thank my advisor Christian Rossow for welcoming me into his group as a PhD student many years ago and his mentoring on the journey that ensued. During this time, Christian always had an open door, ear, and mind for my problems, hurdles, and ideas, which helped to keep me on track, but also allowed me to pursue research ideas on whatever sparked my interest. It was this unique mixture of freedom and guidance that I very much enjoyed during my time as a PhD student in his group.

I would also like to thank my examination committee. Andreas Zeller and Katsunari Yoshioka for reviewing my thesis, Raimund Seidel for agreeing to chair my defense (even on short notice), Ahmad Ibrahim for keeping protocol, and all of them for their interesting and interested questions during the defense.

A special thanks to Giorgi and Michael, who I was fortunate to share an office (and stash of chocolate) with, as well as to Giancarlo, Markus, Jonas, Fabian, Benedikt, Ahmad, Gaganjeet, and Ilya, the other former and current members of Christian's group (in chronological order) I had the pleasure to meet and work with. Thank you not only for your excellent expertise in a wide range of topics and many insightful discussions in front of various whiteboards, but also for the many good memories from our retreats and Christmas parties (from karting to soldering). I would also like to thank everyone else at CISPA for making my time as enjoyable as it was, both abroad at conferences and at home in pre-deadline pizza sessions.

I must further also thank my co-authors and all the other people outside of CISPA I had the privilege to work with. In particular, I would like to thank Mohammad Karami and Damon McCoy for our paper on attack attribution to booter services; Olivier van der Toorn, Mattijs Jonker, Roland van Rijswijk-Deij, and Anna Sperotto for their interesting collaboration ideas and teaching me some intricacies of DNS; Nico Döttling, Dario Fiore, Nils Fleischhacker, Dominique Schröder, and Mark Simkin for letting me have a short stint into the realms of cryptography; Ethan Katz-Bassett and Italo Cunha for pointing out an uncovered edge-case in an early version of our work on BGP-based traceback; and Leyla Bilge and her amazing team at Norton Research Group in Sophia-Antipolis for making my internship a great experience, on both a technical and a personal level.

Finally, I want to thank my friends and family for their continued support, for letting me freely vent about reviewer B, but more importantly, for always reminding me that there is a life outside of work to be enjoyed: Carolyn, Claudia, Curd, Fabian, Frank, Jeanette, Jenni, Michel, Oliver, and Pascal. And last, but definitely not least, thank you Lisa, for being you and enduring me, even through these troubling times.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

The Internet is without a doubt one of the most disruptive innovations. In fact, modern life depends in almost all facets on interconnected systems and services, from interpersonal communication and leisure activities like gaming, to banking, online businesses, and industrial control systems. Unfortunately, this also means that attackers can cause great harm simply by disrupting access to such services. As such, Denial-of-Service (DoS) attacks have reportedly been used to harm competitors [77], as a form of protest by hacktivists [93, 4], as part of extortion campaigns [34], or as means to prevent protesters from organizing by blocking their communication [150]. Furthermore, DoS attacks are no singular incidents. A recent report by NetScout reports more than 10 million DoS attacks in 2020, with many service providers ranking DoS attacks as one of the biggest threats [100].

In recent years, amplification DDoS attacks have emerged as one of the most threatening types of Denial-of-Service. As a subclass of volumetric, distributed DoS attacks, they attempt to render a target inaccessible by flooding it with vast amounts of unwanted traffic, eventually leading to a network congestion. Not only are amplification attacks plentiful nowadays, they are also responsible for many of the most powerful attacks we have ever seen, reaching attack bandwidths on the order of terabits per second [99, 9].

In an amplification attack, the attacker tricks innocent servers into becoming traffic reflectors, by sending them spoofed requests that seemingly originate from the victim. By simply responding to those perceived benign requests, these innocent servers become involuntary accomplices in the attack, sending unrequested traffic to the victim. Furthermore, by picking servers whose responses are multiple times larger than the requests required to provoke them, the attacker gains the eponymous bandwidth amplification, often by a factor of 100× or more [120]. In addition, amplification attacks are also enticing as they inherently hide their attack source: The victim only receives traffic from the involuntary reflectors, and the reflectors themselves can only see spoofed requests.

Today, efforts to mitigate the threat of amplification DDoS attacks are largely focused on reducing the number of vulnerable systems prone to amplification, which has shown promising results in the past [80, 86]. However, it is clear that the sheer scale of the Internet makes a 100% remediation impossible. Likewise, replacing the IP protocol with a secure alternative to tackle the underlying problem of IP spoofing would require a major redesign of the core Internet architecture and must hence be regarded infeasible. Therefore, measures that aim at making amplification attacks *impossible* on a technical level alone are insufficient.

Ideally, they would thus be complemented by measures that make such attacks *unviable* to malicious actors by raising the associated costs and risks. The main cost for an attacker is maintaining a spoofing-capable infrastructure, and given that DDoS attacks are illegal in many jurisdictions, their main risk is legal prosecutions. Yet, actions like seizure of attack infrastructure or launching criminal investigations require forensic evidence of the true network source behind an attack—a non-trivial feat for amplification attacks, as we still lack practical solutions for finding the origin of IP spoofed traffic.

Past proposals rely on either packet marking [140, 45, 43, 132, 14, 54], where path information is encoded in the packet header by individual routers, or correlating traffic statistics captured from routers at a large number of ISPs [137, 138, 87, 75, 147]. However, although both work perfectly in theory, their deployment requirements have proven prohibitive in practice. Packet marking needs to be explicitly supported by all routers along a path due to the necessary protocol changes, while telemetry-based approaches require the cooperation of multiple network providers to collect and share flow data.

The first research question of this thesis is therefore: **(RQ1) How can we provide practical network origin traceback for amplification attacks that does *not* require the cooperation of multiple external parties nor changes to well-established Internet protocols?** After introducing the necessary background in Chapter 2 and reviewing related work in Chapter 3, we answer this question in Chapters 4 to 6, where we propose three traceback approaches that make different tradeoffs between the range of attacks they can be applied to, extra knowledge required about the attacker a priori, and the resulting traceback granularity.

Specifically, in Chapter 4, we start from the assumption that attackers might reuse the same infrastructure for both, scanning for vulnerable systems and launching attacks. We then show how a fingerprint can be imposed on the scanner's IP which is preserved in later attacks and devise a methodology to check whether scan and attack are launched from the same network. Combined, this provides us with an approach to find the IP address of the scanner and, for attacks that are launched from the same network, also the origin network.

Next, in Chapter 5, we consider the setting where the origin is known for some attacks and we would like to know if further attacks can be attributed to that origin. This scenario is motivated primarily by booter services, who essentially provide DDoS-for-hire capabilities and are assumed to be responsible for a large share of attacks [71, 123, 124, 72]. Here, we collect a ground truth dataset by requesting attacks against ourselves from a number of booters and then show how a classifier can be trained on characteristic attack features.

Lastly, in Chapter 6, we study the relaxed traceback problem of finding the autonomous system (AS) that is originating attack traffic. Since the attacker and AS operator must have some form of business relation, the originating AS can then serve as a good starting point for further investigations. Additionally, providing evidence that an AS is emitting spoofed attack traffic allows its peers to pressure the spoofing AS into deploying proper egress filtering.

Crucially, all of our traceback approaches can be deployed by a single entity. To this end, we leverage honeypot reflectors to monitor and attract attack traffic. These honeypots mimic systems that offer multiple protocols suffering from amplification vulnerabilities and can be found by attackers through scanning. Consequently though, our traceback efforts are ultimately limited to the attacks these honeypots can observe. Yet, creating the necessary protocol emulations is a tedious process in which an analyst needs to determine the necessary request-response behavior for a given vulnerability. Likewise, finding these vulnerabilities in the first place requires substantial manual effort, where oftentimes new vulnerabilities become known only *after* they are abused in attacks. The defensive side thus lags behind and is limited to a reactive course of action.

Our second research question is thus: **(RQ2) How can we find amplification vectors and update defensive tooling *proactively*?** We provide an answer in Chapter 7, where we show the first systematic approach to finding amplification vulnerabilities through fuzzing and develop an automated pipeline that synthesizes corresponding honeypot abstractions through symbolic execution.

Contributions

The following gives a brief overview over the publications included in this dissertation [P1, P2, P3, P4] and how they relate.

Identifying the Scan and Attack Infrastructures Behind Amplification DDoS Attacks

In “Identifying the Scan and Attack Infrastructures Behind Amplification DDoS Attacks” [P1] (see Chapter 4), we show a technique for mapping amplification attacks back to the system that was used to scan for reflectors in preparation for the attack. For this we extend an amplification honeypot with a technique we dub *Selective Response*: This special honeypot has 48 public IPs, but only responds to requests on a subset of 24 of those. However, the presented subset differs depending on the requesting IP address and protocol. This ensures that the subset of honeypot IPs a given scanner can find is virtually unique and can act as the scanner’s fingerprint. We can then link attacks to scanners by correlating the honeypot IPs involved in an attack with the scanner fingerprints. This works even for attacks that use only some of the 24 possible fingerprint IPs. As the fingerprints are distributed randomly we can further assign a confidence score to the attribution, based on the likelihood of a random match between the attack and a scanner. Analyzing 1.35M attacks we find that for 58% we can find a unique scanner with a confidence of 99.9% or higher.

In a second step we analyze whether the system used for scanning and the system used to launch the attack are related. To this end, we consider the time-to-live (TTL) values found in packet headers. When forwarding packets, every hop along the route will decrement this value by one. For a fixed initial TTL, the TTL value recorded at the honeypot thus provides a rough estimate of the (network) distance between the sender at the honeypot. Using a network of globally distributed honeypots allows us to obtain multiple distance measurements, which we can then compare between scan and attack. After validating this technique by repeatedly measuring packets from known locations using the RIPE Atlas system [117], we identify 34 scan sources that are also likely used for attacking.

Linking Amplification DDoS Attacks to Booter Services

Our second paper, “Linking Amplification DDoS Attacks to Booter Services” [P2] (see Chapter 5), focuses on attacks launched by booter services. Booter or stresser services provide DDoS-for-hire capabilities, often under the guise of “network stress testing services”, and are believed to be a major source of DDoS attacks. After searching for booter advertisements on online market places and underground forums, we launched attacks from 23 booter services against ourselves, in order to find distinguishing attack characteristics between the services when observed at the honeypot. We find that the subset of honeypots involved in attacks, the number of unique source ports, as well as the distribution of TTL values serve as characteristic features, on which we can train a classifier. To this end, we define dissimilarity functions over the three features, which allows us to compute a “distance” between two attacks and enables us to use a k-nearest-neighbor (k-NN) classifier. As it is impossible to obtain an exhaustive list of possible attack sources, we further introduce a threshold to the classifier, beyond which no attribution is made, denoted by the special label `unknown`.

We then evaluate our classifier in three settings: First, a classical cross-validation ensures that the classifier performs well for all booter services. Second, in a leave-one-out scenario we train the classifier on the attacks from all but one booter service and check whether attacks from the remaining service are correctly classified as `unknown`. Lastly, to assess the stability of features over time, we perform a "real-time" validation: The classifier is asked to classify an attack after being trained on all prior attacks. Although our classifier achieves a high precision of over 99% in all scenarios, its recall drops just below 70% in the real-time scenario, indicating that booters regularly update their reflector set to combat churn.

The original publication [P2] further contains an evaluation of a similar classifier trained on data observed by the *victim* of an attack, which was provided by Mohammad Karami and is hence *not* part of this dissertation.

BGPeek-a-Boo: Active BGP-based Traceback for Amplification DDoS Attacks

While our first traceback approach can find attack sources only if they are also used for scans and our second approach is limited to attacks from booter services which have been observed before, our third paper, "BGPeek-a-Boo: Active BGP-based Traceback for Amplification DDoS Attacks" [P3] (see Chapter 6) provides a coarse-grained, yet general attack traceback mechanism. Specifically, as the Internet is a network of smaller networks, so-called Autonomous Systems (ASes), we relax the attribution problem to finding the autonomous system the attack originates from. ASes relay routing information from one another via the Border Gateway Protocol (BGP). By crafting special BGP announcements, certain ASes can be excluded from receiving a route, a technique known as BGP Poisoning. BGPeek-a-Boo leverages the fact that packets sent by an attacker, whether spoofed or not, can only follow routes that are available to the attacker's AS. Through BGP Poisoning, we can thus restrict the set of ASes from which attack traffic may by sent to our honeypot. By correlating the presence or absence of attack traffic with the set of ASes currently restricted we can infer which ASes are originating or forwarding attack traffic.

We present two traceback algorithms based on this observation. Our first algorithm performs a naive linear scan over the entire AS space and shows the general feasibility of BGP-based traceback, with an estimated median runtime of 91.5 hours. Our second algorithm improves this by over 80% to 16.4 hours when provided with a graph model of AS relationships. This graph model enables it to prune large parts of the search space.

We evaluate both algorithms in a custom simulator modeling over 65k ASes and support our simulation parameters with real-world experiments, where BGPeek-a-Boo achieves a unique traceback result 60% of the time.

Our original publication on BGPeek-a-Boo [P3] won a distinguished paper award at the IEEE EuroS&P 2021 conference.

AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities (and Using Those to Synthesize DDoS Honeypots)

All of our three traceback approaches rely on honeypots to observe attacks and to collect information. However, creating such honeypots is largely a manual effort, in which a vulnerable protocol or implementation is analyzed by hand in order to create a stub emulation of the relevant request/response behavior. Furthermore, as most new amplification vulnerabilities only become known due to being used in attacks, it is also a large reactive process, with attackers one step ahead. Our final paper, "AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities (and Using Those to Synthesize DDoS Honeypots)" [P4] (see Chapter 7), therefore proposes a system that automates both steps, finding amplification vulnerabilities in a systematic way and synthesizing honeypots from observed request/response pairs.

In the first part, we propose a specialized fuzzer, built on top of the state-of-the-art guided greybox fuzzers ParmeSan [102] and Angora [25]. For finding amplification vulnerabilities, we use a combination of static analysis and dynamic instrumentation to make the AmpFuzz *UDP-aware*. This enables the fuzzer to know when the service under test expects input packets and when a request has been processed, allowing for a thousandfold increase in fuzz-throughput compared to timeout-based solutions. We further guide the fuzzer towards sending network functions and leverage Angora's byte-level taint tracking to give fuzzing priority to input bytes affecting the size of response packets. In an evaluation on 25 network services from the Debian repositories we (re-)discover five known and four previously unreported amplification vulnerabilities.

For honeypot synthesis, in the second part we instrument our fuzz targets with a symbolic execution engine, SymCC [111]. This enables us to replay the identified amplification requests concolically, which provides us with symbolic conditions that the request has to satisfy as well as symbolic expressions of all output bytes of the generated response. We then translate these constraints and expressions into Python code, namely one function for checking the request and one function for computing the response for every request/response pair. These generated functions can then serve as the protocol emulation layer in DDoS honeypots.

Outline

The remainder of this dissertation is structured as follows. Chapter 2 recapitulates the technical background on amplification DDoS attacks and introduces the AmpPot honeypot, which serves as the basis for our traceback approaches. In Chapter 3 we review related work pertaining to all chapters, namely from the areas of (amplification) DDoS and existing traceback approaches. Where applicable, later chapters will include a dedicated review of specific related work. Our three traceback approaches are presented in Chapters 4, 5, and 6, and AmpFuzz in Chapter 7. Finally, Chapter 8 concludes this dissertation.

2 Background

In this chapter we provide a brief description of amplification DDoS attacks and their underlying mechanism and introduce AmpPot, a honeypot tailored towards monitoring such attacks.

2.1 Amplification DDoS Attacks

Amplification attacks have become one of the most popular and dangerous classes of Distributed Denial-of-Service (DDoS) attacks nowadays. Denial-of-Service (DoS) attacks aim to render a target system unusable. In the case of amplification DDoS attacks this is achieved by flooding the target with large amounts of network traffic, eventually exhausting all available bandwidth. To this end, the attacker tricks public UDP services (e.g., DNS servers) into sending large amounts of traffic to the victim. Given the connection-less nature of UDP, an attacker can steer the services' responses to the victim by simply spoofing the source IP address in requests. The service will then perceive this packet as a legitimate request and respond to the victim (making the service an involuntary *reflector*). Attackers abuse up to thousands of reflectors [124] to *distribute* the attack traffic and complicate defenses on the victim side. By further carefully selecting reflectors that send large responses, the attacker can maximize the traffic that is reflected to the victim and achieve traffic *amplification* [97].

Several protocols have been identified to be amplification-prone [120], with amplification factors (ratio between response and request size) ranging from 5 to 4 000. In addition, misconfigurations and support for legacy protocol options lead to a plethora of potential amplifiers. As a result, amplification attacks constitute a large fraction of high-bandwidth DoS attacks. Successful attacks have temporarily disrupted core Internet services, such as Spamhaus (in 2013 [113]) or Paypal, Spotify, Twitter, Reddit, and eBay (all in 2016 [76]), with reported attack bandwidths as high as 1.7 Tbps in 2018 [99] or 2.3 Tbps in 2020 [9].

However, amplification DDoS attacks not only enable attackers to reach very high bandwidths with little effort, they also provide a layer of anonymity due to their inherent reliance on IP spoofing. In an attack, all packets received by the victim originate from the (innocent and involuntary) amplifiers, while all packets received by the amplifiers carry the spoofed source address of the victim. This makes attack attribution, and thus also criminal prosecution, highly non-trivial—a problem which motivated large parts of this thesis.

2.2 AmpPot

AmpPot [S1] is a honeypot that allows to observe and monitor DDoS amplification attacks. To this end, AmpPot mimics a server running several protocols vulnerable to amplification and aims to be selected as an amplifier during attacks. During an attack, the attacker will then also send spoofed requests to AmpPot, which can then record the attack, the IP address of the victim (the request's source address), and further protocol specific information, such as the domain names abused in DNS-based attacks. In order not to contribute to the actual attack traffic, AmpPot employs a strict rate-limiting, answering at most three requests at a rate of one request per minute. /24 networks exceeding this threshold are blocked for one hour. This ensures that harm towards attack victims is minimized, while still enabling attackers to find AmpPot instances when scanning for potential reflectors.

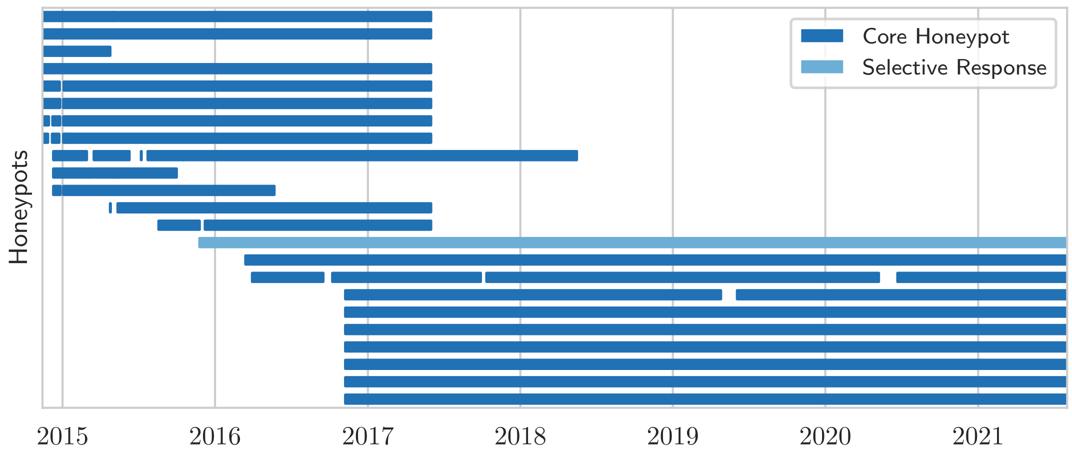


Figure 2.1: Timeline of AmpPot deployment

While Rossow [120] had originally used full service implementations to bait attackers and collect attack information, AmpPot uses only lightweight protocol emulations, which greatly reduces the honeypots complexity and computation requirements. For protocol emulation, incoming requests are checked against patterns of known amplification requests, while responses can be either fully hardcoded or generated from templates (e.g., substituting request IDs or timestamps). Only for DNS, where responses depend on the queried domain name, a caching recursive resolver is used to obtain the corresponding DNS records once.

All defense mechanisms presented in the following chapters rely on AmpPot in one way or another. As such, AmpPot saw continuous development during the course of this thesis, most notably with the Selective Response extension (see Section 4.6), but also with adding support for additional protocols and a major architecture re-design.

Figure 2.1 shows a timeline of AmpPot installations. The initial AmpPot deployment consisted of eleven honeypots installed towards the end of 2014, supporting six vulnerable protocols: QOTD (17), CharGen (19), DNS (53), NTP (123), MSSQL (1434), and SSDP (1900). In 2015, two of these were replaced and towards the end of the year a special Selective Response honeypot was added to the network. In 2016, it became clear that the original AmpPot deployment had to be abandoned and a new network was established, starting with two instances and scaling to nine later. Furthermore, support for TFTP (69) was added. As expected, in 2017, most of the original AmpPot instances had to finally be shut down, with only a single honeypot surviving until mid-2018. Following reports of attacks abusing Memcached [31], support for the Memcached (11211) protocol was added in early March 2018, and LDAP (389) and RCPBIND (111) followed in 2019. Since then, the AmpPot installation encompasses nine core and a Selective Response honeypot, listening on a total of 57 IP addresses.

Due to the vast amount of traffic in a DDoS attack, AmpPot employs sampling approach to log incoming packets: Once a source exceeds 100 packets within one hour, packets from this source are only recorded with a probability of 1/100. To distinguish attacks from scans

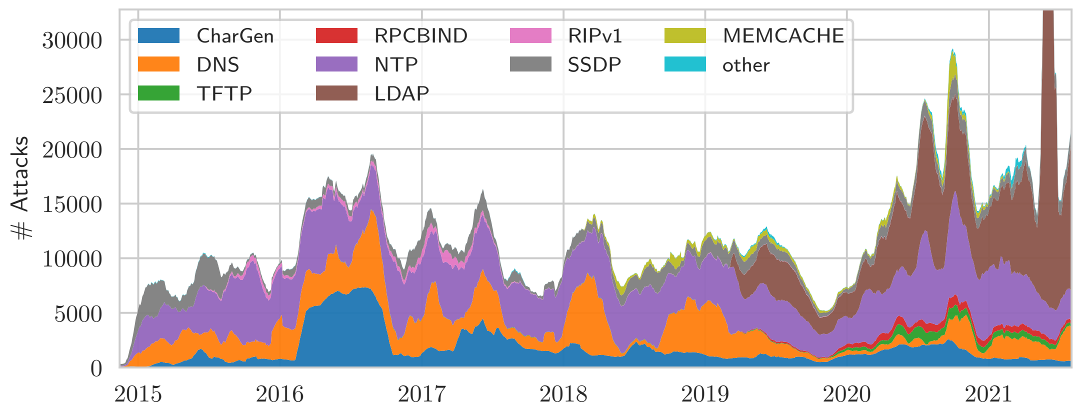


Figure 2.2: Attacks observed by AmpPot

and other noise, AmpPot defines an attack as a stream of at least 100 consecutive packets from the same source to the same port with no gaps longer than one hour.

Figure 2.2 shows the number of attacks recorded per day and protocol (averaged over 4 week periods for legibility). On average, AmpPot observes around 10k attacks per day, which highlights the prevalence of amplification attacks. The share of NTP-based attacks has been relatively constant over the years, which, for the most part, also holds true for DNS. CharGen saw high popularity in 2016, but its share has declined since, while the majority of attacks observed by AmpPot nowadays leverages LDAP, which has seen increasing popularity since its addition to the honeypot in 2019.

3
Related Work

Since they form the overarching theme, in this chapter we review existing works from the areas of amplification attacks and traceback mechanisms. Work from other topics alluded to by individual chapters will be discussed therein.

3.1 Amplification DDoS

The idea of using third-party services as reflectors for DDoS attacks was first brought forward by Paxson in 2001 [104], showing the UDP-based reflection potential of DNS and SNMP. In 2014, Rossow extended the list of known-vulnerable UDP protocols to a total of 14 and provided a measurement of their real-world amplification factors [120].

3.1.1 Amplification Vectors

Following that, a number of works have further analyzed individual protocols for their amplification potential: For DNS, van Rijswijk-Deij et al. studied the impact of the then-newly introduced DNSSEC [116], while MacFarland et al. analyzed how an attacker can optimize their queries to achieve larger amplification factors [91]. The works of Sun et al., Liu et al., and Adamsky et al. all show how peer-to-peer networks can be leveraged to launch amplification attacks [145, 146, 88, 2], including a scenario in which the attacker first uploads data to a distributed storage system and later spoofs download requests from the victim. The latter attack is conceptually similar to the Memcached amplification attack [31] observed in the wild in 2018. Beyond UDP reflection, Kührer et al. investigate the amplification potential of the TCP handshake itself [81], Sargent et al. that of the IGMP management protocol [125], while Gasser et al. warn about the threat posed by publicly reachable BACnet devices [56]. Lastly, Moon et al. [97] present a global scanning service AmpMap, which uses blackbox fuzzing to probe servers on the Internet for their amplification potential.

3.1.2 Defense Mechanisms

Knowing amplification vulnerabilities is vital for a number of reasons. For one, it allows to assess the threat landscape by scanning for potential amplifiers or monitoring (malicious) scanning activities using network telescopes. Using these techniques, Czyz et al. provided an early measurement of NTP DDoS attacks in 2014 [33] where they combine active scanning and network telescope data with the views from a DDoS mitigation provider and network flow data from multiple ISPs. They further leverage a peculiarity of the NTP `monlist` amplification, whereby the victim's address and target port are returned on subsequent `monlist` requests in order to characterize the victims of such DDoS attacks, finding that a large number of victim services are gaming related. In a similar vein, in 2015 Kührer et al. studied the population of open DNS resolvers through Internet-wide scans over a 13 month period [79]. Furthermore, once a vulnerability is known, steps can be taken to mitigate it. Both of the studies above find the total number of available amplifiers declining over time. A 2014 study by Kührer et al. [80] and a 2016 study by Li et al. [86] further explore the impact of active vulnerability notifications to operators of DDoS amplifiers. While Kührer et al. report a reduction of vulnerable NTP servers by 92% achieved through a coordinated disclosure in collaboration with multiple NOCs and CERTs, the later study by

Li et al. shows only small remediation effects on other protocols, even when contacting the operators directly.

Next to defending against amplification attacks by reducing the number of reflectors, others have also developed mechanisms to detect and defend against amplification attacks and volumetric DDoS in general: For example, Wang and Reiter propose to embed a proof-of-work-like mechanism at the IP layer in order to make volumetric attack infeasible [161], Kreibich et al. attempt to detect and drop DDoS traffic due to its inherent asymmetry [78], while Gilad et al. show how IaaS cloud providers can be used to create impromptu CDNs once an attack is detected [58]. Towards monitoring attacks, in 2017, Thomas et al. also used a honeypot, HopScotch, similar to AmpPot, for a long-term analysis of the amplification DDoS landscape [153], while Jonker et al. provide different perspectives on the DDoS ecosystem by fusing AmpPot data with other data sources such as network telescopes.

3.1.3 Booter Services

Since they are suspected to be a major source for amplification attacks, several researchers have also analyzed booter services: For example, Santanna et al. analyzed 14 booter services and the attack types they offer [124]. In another paper, Santanna et al. obtained and studied attack logs from 15 booter services [123], revealing insights into the distribution of attack victims and the booter website hosters. They also investigated booter "infrastructures", but focused on the web front-ends of the booter services—which are decoupled from the booter attack infrastructure and much easier to replace than the infrastructure that we aimed to trace back. Karami et al. document this clear separation between websites and back-end servers [71, 72]. They study the ecosystem of three booter services, show the difficulty and costs for renting such back-end servers, and analyze two "spoofing-friendly" hosting providers. However, none of these studies have investigated ways to identify the actual attack infrastructures or link attacks back to booter services.

3.2 IP spoofing and Traceback

IP spoofing and the resulting need for traceback [70] has been an active field of research for many years. While concepts for closing the root cause of amplification attacks (IP spoofing) are well-known [48, 11, 141], little success has been made in *identifying* the spoofing sources. One common approach collects (statistical) telemetry data at multiple routers from which the origin of spoofed packets can later be derived [137, 138, 87, 147, 75]. Another approach lets routers encode path information into the packet itself [127, 42], using additional IP options or unused header fields [127, 140, 35, 165, 26], advanced encoding schemes [140, 164, 54], or probabilistic techniques [127, 126, 45, 43, 132] to reduce the per-router overhead. In theory, both approaches could perfectly track the origin of spoofed traffic. However, both require a widespread deployment in routers and the cooperation of multiple ISPs. As some of these techniques have been proposed almost two decades ago, it is clear that this is an inhibiting factor for traceback in practice. A third line of research considers the broader question of identifying systems that are *capable* of IP spoofing [16, 17, 80, 89, 23].

4 Identifying the Scan and Attack Infrastructures Behind Amplification DDoS Attacks

4.1 Motivation

Amplification DDoS attacks have gained popularity and become a serious threat to Internet participants. However, little is known about where these attacks originate, and revealing the attack sources is a non-trivial problem due to the spoofed nature of the traffic. In this chapter, we present novel techniques to uncover the infrastructures behind amplification DDoS attacks. We follow a two-step approach to tackle this challenge: First, we develop a methodology to impose a fingerprint on scanners that perform the reconnaissance for amplification attacks that allows us to link subsequent attacks back to the scanner. Our methodology attributes over 58% of attacks to a scanner with a confidence of over 99.9%. Second, we use Time-to-Live-based trilateration techniques to map scanners to the actual infrastructures launching the attacks. Using this technique, we identify 34 networks as being the source for amplification attacks at 98% certainty.

4.2 Problem Description

Due to the IP-spoofing nature of amplification attacks, the true origin of the attacks remains hidden. Consequently, victims do not know whom to contact to stop the attacks, nor can they file legal complaints against attack originators. Even worse, from the victim's perspective, the third-party reflectors may appear to be the attack origin, giving false attribution hints. Despite the need for effective mechanisms to trace back the origin of amplification attacks, we still lack usable mechanisms. While there are attempts to identify spoofing-enabled networks [23, 80], the coverage of such active probes is limited, and without further evidence of abuses, the identified parties feel little social pressure to ban spoofing from their networks (e.g., using BCP38 [48]).

4.3 Contributions

In this chapter, we tackle this problem and aim to attribute amplification attacks back to the infrastructures that have caused them. Whereas application-layer DoS attacks can be attributed to the origin due to the nature of the TCP handshake, finding the source of amplification attacks is inherently more difficult, given that attackers (i) spoof the IP addresses and (ii) use reflectors to diversify traffic sources. IP traceback and similar packet marking schemes have long been the *de facto* standard proposal to detect the origin of spoofed traffic [127, 42, 140, 166, 54, 43, 45], but none of these designs were ever deployed at scale to allow for global attack attribution. Still, decades after spoofing attacks were discovered, tracking the origin of attack traffic remains an ad-hoc process that requires coordination between many ISPs scattered around the world. The outcome of such tedious manual attribution processes (if any) comes hours or days after the traffic originated.

We follow a two-step process to establish an attribution process that identifies the infrastructures operated by attackers to *prepare* and *launch* amplification attacks. In a first step, we aim to link the *reconnaissance* and the *attack* phases by tracking which scan for amplifiers has resulted in which attacks. We leverage the fact that scans cannot forge their source IP address and thus learn about the scanning infrastructures, despite the fact that attack traffic is spoofed. Our key idea is to offer each scanner a different set of potential honeypots that it can abuse. This way, we implicitly encode a secret identifier to the set of honeypots that any subsequent attack will use, which varies per scan source. In a second step, we test if the scan infrastructure is also used to actually *launch* (and not just to prepare) the attacks. We follow the observation that similar traffic sources should have similar "distances" (in terms of hops) to globally-distributed sensors. Using trilateration, we can link scanners to attack origins based on hop counts.

With these proposals, we provide a practically-usable attribution methodology for amplification attacks. Our framework fulfills important goals: (i) Our method can work in real-time; that is, we can attribute attacks on the fly without noticeable delay between attack start and attribution outcome. (ii) The attribution does not require any cooperation between ISPs, and thus solves one of the main practical problems of existing solutions like IP traceback. (iii) Our method gives probabilistic guarantees that show if—and at what confidence level—the attribution outcome is correct.

We have deployed our attribution methodology on a snapshot of 1 351 852 amplification attacks monitored by honeypots during 23 weeks in 2015 and 2016. Our findings show that we can identify the scanners that were used during the reconnaissance phase of 58% of all attacks in our data set. Further analyses show that only 20 scanners are responsible for nearly 50% of the attacks. Using our hop-based trilateration process, we reveal that 22% of the attacks were actually launched from scan infrastructures, for which we have perfect IP, network and geographical attribution information. We report on the distribution of attack sources and reveal black sheep networks that cause massive spoofing attacks.

To summarize, our contributions are as follows:

- We present a novel honeypot-based technique, *selective response*, that enables us to assign a fingerprint to scanners during the reconnaissance for amplification DDoS attacks and give confidence guarantees for subsequent attribution.
- We evaluate our technique on a set of 1 531 852 attacks recorded by our honeypot, of which we can link 785 285 back to their corresponding scanner with a confidence of 99.9% or higher.
- We leverage the TTL field of the IPv4 header to compare the location of scanners to origins of attacks, after evaluating our methodology on data collected by RIPE Atlas probes.
- We find that for 22% of all attacks, the scanner linked to the attacks is also the source of the attack with 95% confidence.

The remainder of this chapter is structured as follows. In Section 4.4, we define our threat model, discuss the ethical implications of our work, and describe the data used in this chapter in Section 4.5. Section 4.6 introduces a novel honeypot-based technique to assign identifiers to systems that scan for amplifiers. We evaluate this technique in Section 4.7. In Section 4.8, we measure if the infrastructure used to scan for amplifiers is identical to the infrastructure used to launch amplification DDoS attacks. After reviewing our initial assumptions in Section 4.9 and providing an overview of related work in Section 4.10, we conclude with Section 4.11.

4.4 Background

In this section, we will define the threat model this chapter considers and discuss the ethical implications of our work.

4.4.1 Threat Model

The focus of this chapter is on amplification DDoS attacks. Our threat model is therefore comprised of at least three parties: The *attacker*, the *victim*, and a set of *amplifiers*. Finding these amplifiers is a vital step in attack reconnaissance, and is typically performed by scanning, i.e., sending requests to every address in a given range, and recording the answers. Therefore, we include this additional fourth party of a *scanner* to our threat model.

While attackers could potentially leverage botnets to launch amplification attacks, previous work documented that the vast majority of amplification attacks stem from a single origin [S1], which also coincides with our findings. In the following, we will thus assume that attackers use only a single system to launch their attacks.

We will further assume that scanners do not spoof their source addresses when performing a scan. While techniques to perform scans using spoofed addresses are known for TCP (e.g. "idle scan" [122]), no similar techniques are known for UDP. Since all known amplification DDoS attacks are UDP-based, this is a valid assumption.

4.4.2 Ethical Considerations

The data sets used in this chapter were collected using a modified version of AmpPot [S1]. Deployment of such a honeypot poses a challenge from an ethical point of view: By design, an amplification honeypot will also act as an amplifier in an actual attack and thus send unwanted traffic towards DDoS victims.

We argue that the contribution towards attack traffic by our honeypot is negligible and only incurs minimal harm to the victim's system. We did not modify the default thresholds of AmpPot, by which the honeypot will answer at most three requests per attack. Although we deployed our honeypot to listen on 48 IPs, as discussed later, at most 24 of those IPs will send replies towards a victim's system. Therefore, our honeypot will reply to at most 72 packets in total, i.e., a few kilobytes at most. Taking into account that these attacks usually flood a victim's system with traffic in the order of several Gbit/s, we conclude that the contribution of our honeypot is negligible.

In addition, we offered attack victims a method to opt out from our measurements. During the course of our experiments, we received three complaints that we immediately answered describing our experimental setup, but none of the complainers asked to opt out. We refer the interested reader to a more detailed ethical discussion in [S1].

4.5 Dataset

For this chapter, we leveraged AmpPot in two ways:

First, to attribute attacks to scanners, we extended AmpPot in that it only *selectively* replies to requests. The basic idea behind this is that every scanner will see a different set of honeypots, which will become a distinctive feature for attribution. We provide an in-depth description of this technique in Section 4.6. We deployed our modified AmpPot version on Nov. 25th, 2015.

Second, we also use data collected by 11 unmodified honeypots which were deployed in late 2014 and have been operated since then. Combining their view with that of our modified honeypot allowed us to examine whether scans and attacks were launched from the same infrastructure by comparing TTL values (cf. Section 4.8).

We base our results on data collected between November 25th 2015 and May 1st 2016 (exclusive). Within this time our modified AmpPot version observed 1 351 852 attacks, 1 254 102 of which were also observed by some of the unmodified AmpPot instances.

4.6 Selective Response

In this section, we describe the idea that our honeypots selectively respond depending on the scanner origin—a fundamental technique that allows for attack attribution.

4.6.1 Intuition

Launching amplification attacks requires prior knowledge of a set of servers which can be abused as amplifiers during the attack. Finding such servers is commonly achieved through scanning, i.e., sending a query to every IP in a certain range, and recording which IPs send back a reply. Since nowadays scanning the entire IPv4 address space is feasible in a reasonable amount of time even from a single machine [44, 63], we assumed that in most cases the chosen amplifier set was based on the scan result(s) from a single scan system. Note that we will verify this assumption in later analyses.

The main goal of this chapter is to correlate scan events with amplification attacks. We therefore follow the idea that every scanner should find a different (ideally unique) subset of our deployed honeypots. To this end, we influence the scan result in such a way that we can re-identify the scanner once its scan result (the set of amplifiers) is used in subsequent amplification attacks. Our approach ensures that within a network segment under our control, every scanner finds a *different* set of potential amplifiers. That is, we launch honeypots on all IP addresses on that segment, but only selectively respond to a scanner-derived and therefore unique subset of IP addresses. If our assumption on single-source scans was correct, this would mean that attacks based on different scans would also use different amplifier sets.

4.6.2 Implementation

Technically, we implemented the selective response scheme as follows. We fix a fraction α of the network that responds to a scan, so that every scanner that performs a full scan on the network of size N would see replies from $\alpha \cdot N$ hosts. In order to maximize the number of possible combinations $\binom{N}{\alpha N}$, we set $\alpha = 1/2$, i.e., respond with exactly half of the honeypots, and remain "quiet" with the other half.

To select the αN IP addresses from the network, we compute a hash over the source IP address (i.e., an identifier for the scanner), the protocol, the base of the network, and a secret key (string). We added the secret key such that an attacker cannot precompute the set of responding honeypots based on our (otherwise deterministic) hash function. The resulting hash is then used to derive a permutation of the N IP addresses in the network. From this permutation, the first αN addresses are selected as responding honeypots.

Our selective response honeypot uses three /28 networks, which gives a total of 48 static IPs distributed over three networks with 16 IP addresses each. Due to the split over three distinct /28 networks, we decided to perform the selection *per subnet*. Separating the networks into three independent ranges has the advantage that we can even attribute scanners that scanned only one of the networks. Although this separation reduces the number of possible combinations from $\binom{48}{24} \approx 3.2 \times 10^{13}$ to $\binom{16}{8}^3 \approx 2.1 \times 10^{12}$, it is still two orders of magnitude larger than the number of IPv4 addresses, i.e., it is very likely that every scanner will be assigned a unique set of 24 amplifiers.

4.7 Attributing Attacks to Scans

4.7.1 Methodology

The basic idea of our attribution is simple yet effective. For every attack we monitor, we inspect the set of honeypots that were abused for this attack. Typically, attackers leverage multiple amplifiers at the same time, and often also multiple of our honeypots are abused for the same attack. Figure 4.1 shows the cumulated percentage of attacks (y-axis) that have an attack set of at least a given size (x-axis). Over 95% of all attacks use at least four honeypots; 80% use at least 10 of our honeypots simultaneously.

Remember that the amplifier set greatly depends on the scan prior to the attack, for which our selective response scheme has introduced artificial entropy. From the set of honeypots abused in an attack, we therefore aim to derive which scanner has discovered these very same honeypots.

Technically, for every IP of our honeypot, we maintain a set of all sources that discovered this IP as an amplifier, i.e., sent a packet to this IP *and* got back a response. We denote those sources as being *aware* of the corresponding IP. Since we cannot perfectly distinguish attacks from scans, we will consider *every* source that contacted our honeypot, which explicitly includes victims of attacks.

Upon attributing an attack, we first extract the set of IPs used as amplifiers in this attack. Conjecturing that the scanner behind this attack must have scanned all of these IPs and

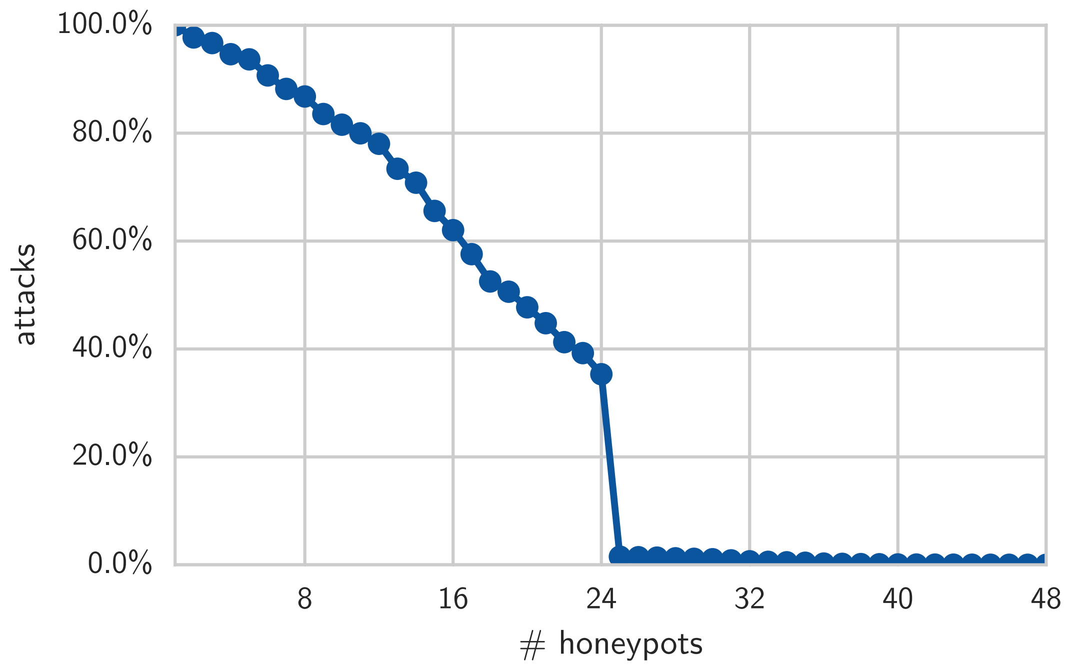


Figure 4.1: Percentages of attacks (y-axis) that use at least the given number of honeypots (x-axis)

honeypot IP	aware sources
10.0.0.1	{169.254.0.10, 192.168.2.100, 198.18.3.24}
10.0.0.2	{169.254.0.10, 172.16.5.27, 198.18.3.24}
10.0.0.3	{192.168.2.100, 172.16.5.27, 198.18.3.24}
10.0.0.4	{169.254.0.10, 172.16.5.27, 192.168.2.100}

Table 4.1: Example honeypots and *aware* scanners

received a reply, it must be contained in the set of aware sources for each of them. We can therefore find the scanner by building the intersection of these sets.

Since neither maintaining the list of aware sources nor computing a set intersection is computationally expensive, our methodology can also be applied in real-time, i.e., once an attack is detected, the result of the attribution can be obtained without any noticeable delay.

Consider the toy example in Table 4.1 that lists the set of aware sources for 4 honeypots. Assume that we observe an attack using honeypot IPs 10.0.0.1, 10.0.0.3, and 10.0.0.4. We can then find the potential scanner in the following way: As 10.0.0.1 is contained in the attack set, the scanner should be one of {169.254.0.10, 192.168.2.100, 198.18.3.24}. Since 10.0.0.3 is contained as well, we can narrow this down to {192.168.2.100, 198.18.3.24}, because 169.254.0.10 is *not* aware of 10.0.0.3. We can likewise exclude 198.18.3.24, as it is not aware of 10.0.0.4. This leaves only {192.168.2.100} as a potential scanner behind the attack.

Mapping scanners this way can result in three cases:

1. **Zero candidates** If the set of candidates is empty, then no single scanner was aware of this set of amplifiers. We will call such attacks *non-attributable*. This can occur for multiple reasons:

   Firstly, it could be that the attack is based on data from multiple scans, in which case the combined amplifier set is likely distinct from the sets found by other scanners. This is especially true for attacks that use more than $\alpha N = 8$ IP addresses per honeypot subnet, as a single scanner can only find up to eight amplifiers in each /28 network.

   Secondly, due to the threshold of 100 packets that determines an attack, a scan can also be mistaken for an attack. If a scanner scans all of our 48 IPs, it can easily exceed the threshold by sending a little over 2 packets per IP. This happens, e.g., for scanners that start with a full scan (i.e., a single packet per IP address) and then verify each responding IP address by sending additional packets.

   Thirdly, because AmpPot considers an attack to have ended only if the packet-rate drops below 100/hour for one hour, two attacks targeting the same source in quick succession can be aggregated into a single attack.

   Although we could neither observe nor refute the first scenario, we have observed both the second and third scenarios in our data.

2. **Exactly one candidate.** If the set of candidates contains exactly one candidate, only a single scanner is aware of this set of amplifiers. We will consider this as a potential attribution. However, since we cannot exclude that the set of amplifiers was chosen by other means (e.g., combining data from multiple scans), we compute a *confidence* for this attribution. The computed confidence gives an indication of how likely it is that this attribution is correct. We give a detailed explanation of the confidence in Section 4.7.2.

3. **More than one candidate.** If the set of candidates contains multiple candidates, multiple scanners are aware of this set of amplifiers. We will call such attacks *non-unique*. This case occurs if the set of chosen amplifiers is relatively small and multiple scanners got responses from those IPs during their scans.

   In this case, we will refine the candidate set by finding scanner-to-victim relations in the set of candidates. That is, if both *A* and *B* are contained in the set of candidates, but have previously observed an attack against *B* which we could attribute to *A*, we will remove *B* from the set of candidates. This is based on the assumption that victims of DDoS attacks will most likely not act as scanners for DDoS attacks themselves, and even if they did, the set of amplifiers found would still be based on *A*'s scan.

4.7.2 Confidence

Even if an attack can uniquely be attributed to a scanner (case 2), it is unclear how confident this mapping is. In an ideal world, a scanner would scan all of our 48 IPs and subsequent attacks would use the full reply set of 24 IPs. Since the full reply set is unique per scanner, this would allow for a perfect attribution. However, in practice, several things impede this ideal-world assumption. For example, scanners might not query all 48 IP addresses. Even if they did, attackers could select a random subset of the found IP addresses to use in attacks. Worse, attackers might not base their attacks on the results of only a single scanner, but rather combine scan results from multiple sources. This raises the question whether our attribution is actually robust under such real-world conditions.

Our approach to answer this question is to define a confidence that expresses how likely it is that our attribution result is actually correct, based on the following two sets. We will call the set of IP addresses that were queried by a scanner the *query set* Q, and refer to the set of IP addresses that replied as the *reply set* R. Since the reply set is determined using a hash function, which we assume to generate uniformly distributed values, we can consider the distribution of reply sets to be uniform as well.

We then analyze the probability with which we would falsely accuse a scanner of being responsible for an attack. The intuition behind this is as follows: Assuming that a given attack was *not* based on the reply set of a single scanner, what is the probability that any of the scanners still matches this attack by chance? That is, what is the probability we falsely accuse a scanner? If this probability is sufficiently small, we can conclude that—if we can attribute this attack to a scanner—this attribution is correct.

Formally, assume an attack that uses the IPs $A = A_1 \sqcup A_2 \sqcup A_3$, where A_i is the set of IPs from the ith subnet. We are now interested in the probability that a scanner that scanned a superset of A also gets replies from all IPs in A, i.e., $\Pr[A \subseteq R \mid A \subseteq Q]$. In each /28 subnet, the reply set the scanner observes is a subset of one out of $\binom{16}{8}$ sets. Out of these, $\binom{16-|A_i|}{8-|A_i|}$ are supersets of A_i (since the A_i IPs from the attack are fixed, a scanner could potentially receive responses from $8-|A_i|$ out of the remaining $16-|A_i|$). Thus, assuming a uniform distribution of reply sets, it holds that

$$\Pr[A_i \subseteq R \mid A \subseteq Q] = \frac{\binom{16-|A_i|}{8-|A_i|}}{\binom{16}{8}}$$

Therefore, the total probability that a scanner that scanned a superset of A also got replies from all IPs in A is

$$p = \Pr[A \subseteq R \mid A \subseteq Q] = \frac{\binom{16-|A_1|}{8-|A_1|}}{\binom{16}{8}} \cdot \frac{\binom{16-|A_2|}{8-|A_2|}}{\binom{16}{8}} \cdot \frac{\binom{16-|A_3|}{8-|A_3|}}{\binom{16}{8}}$$

From this individual probability for a single scanner we can now derive a probability for any scanner in our dataset. The probability that *any* of the S scanners that scanned a superset of A got replies from all IPs in A follows the “at-least-once” semantics and is

$$1-(1-p)^S$$

Put differently, if we can attribute the attack to a scanner, our confidence that this attribution is correct is

$$(1-p)^S$$

For example, assume an attack that uses 5 IPs from the first, 4 IPs from the second, and 6 IPs from the third subnet respectively, i.e., $|A_1| = 5$, $|A_2| = 4$, $|A_3| = 6$. A single scanner then has probability

$$\begin{aligned} p &= \frac{\binom{16-5}{8-5}}{\binom{16}{8}} \cdot \frac{\binom{16-4}{8-4}}{\binom{16}{8}} \cdot \frac{\binom{16-6}{8-6}}{\binom{16}{8}} = \frac{165 \cdot 495 \cdot 45}{12\,870^3} \\ &= \frac{3\,675\,375}{2\,131\,746\,903\,000} \approx 0.0001742\% \end{aligned}$$

of receiving responses from this precise attack set during its scan. If at the time of the attack we had had contact with 200 scanners that scanned our entire network, the probability that *any* of them found this precise attack set is thus

$$\begin{aligned} 1-(1-p)^S &= 1-\left(1-\frac{165 \cdot 495 \cdot 45}{12\,870^3}\right)^{200} \\ &\approx 0.03448\%. \end{aligned}$$

Consequently, if we find a scanner that matches this attack, in 99.966% of all cases this does *not* happen by chance, and hence for such an attack we have a confidence of 99.966% that our attribution is correct.

Obviously, a larger set A will lead to a smaller probability, implying a higher confidence.

4.7.3 Experimental Results

We will now turn to the results of our attribution process using the dataset described in Section 4.5.

Figure 4.2 shows the percentages of attacks that were marked as *attributable*, *non-unique*, and *non-attributable*, respectively. Percentages are given both overall and per protocol. The absolute numbers for each category are given in Table 4.2, as well as attribution results for different levels of confidence. Since the QOTD and MSSQL protocols only account for a negligible number of attacks (1368, $\approx 0.1\%$), we will omit these protocols in the following.

4.7.3.1 Attributable Attacks

Most notably, out of the 1 351 852 attacks that we recorded at our honeypot, 785 285 (58.09%) could be attributed to a single scanner with a confidence of 99.9% or higher. This means that the chance that the attack was *not* based on the attributed scanner is less than 1 in 1 000. In fact, 643 956 attacks (47.64%) even have a confidence of 99.999% or higher, i.e., the chance of a false attribution is less than 1 in 100 000.

Surprisingly, our results are not homogeneous among different protocols. This can be seen in Figure 4.4, which depicts the fraction of attacks that could be attributed for various levels of confidence. While 74.70% of all CharGen attacks could be attributed with a confidence of 99.9% or higher, this holds for only 10.20% of the SSDP-based attacks. This discrepancy stems from the fact that the number of honeypot IPs used strongly varies between protocols, as can be seen in Figure 4.3, which shows the distribution of honeypot IPs used for all

	QOTD	CharGen	DNS	NTP	RIPv1	MSSQL	SSDP	**Total**	
non-attributable	0	1 440	11 428	18 665	40	25	2 460	**34 058**	**(2.52%)**
non-unique	155	24 982	84 491	102 635	191	272	25 046	**237 772**	**(17.59%)**
attributable	53	353 300	230 626	426 962	17 253	863	50 965	**1 080 022**	**(79.89%)**
conf. > 99%	53	294 913	208 696	342 852	15 163	784	13 989	**876 450**	**(64.83%)**
conf. > 99.9%	53	283 665	201 555	279 467	11 928	610	8 007	**785 285**	**(58.09%)**
conf. > 99.99%	53	280 514	179 033	233 914	10 395	536	6 260	**710 705**	**(52.57%)**
conf. > 99.999%	52	274 617	140 055	214 329	8 441	535	5 927	**643 956**	**(47.64%)**
Sum	208	379 722	326 545	548 262	17 484	1 160	78 471	**1 351 852**	**(100.00%)**

Table 4.2: Attribution results and confidence breakdown

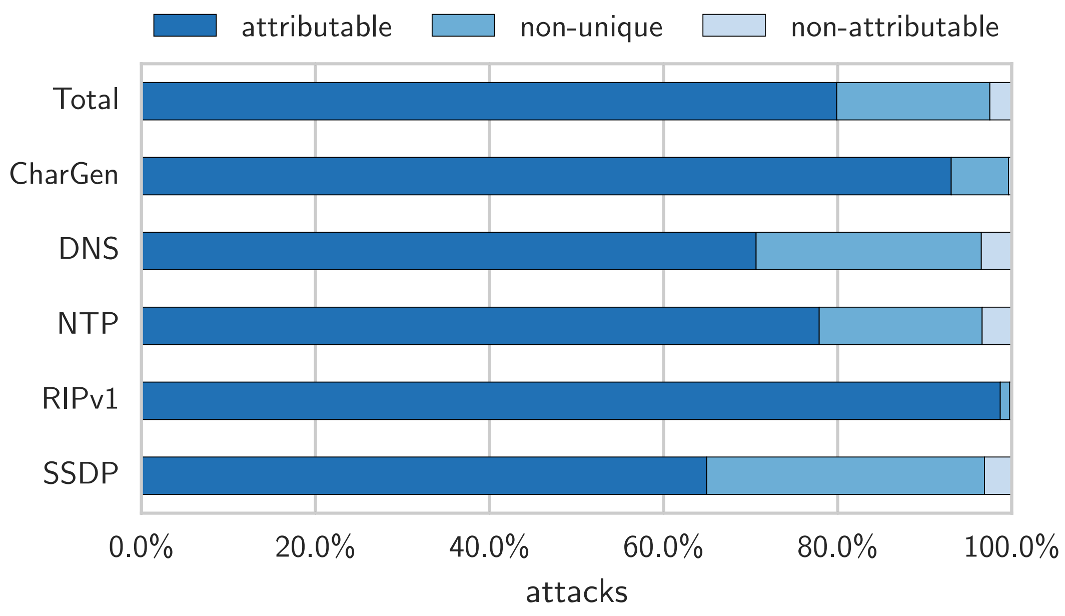


Figure 4.2: Attribution results per protocol

protocols. SSDP attacks only use 9.38 IPs on average, whereas `CharGen` attacks use 20.66. Consequently, SSDP also experiences a higher percentage of attacks marked as non-unique. These discrepancies can be explained by the global number of amplifiers available on the internet. For example, Rossow found 3 704 000 servers vulnerable to be used as SSDP amplifiers, in contrast to only 89 000 for `CharGen` [120]. In other words, our honeypots are less likely to be abused as amplifiers for SSDP-based attacks due to the abundance of available alternative SSDP amplifiers.

4.7.3.2 Non-Attributable Attacks

Only 34 058 attacks (2.52%) were considered to be *non-attributable*. This indicates that our initial assumption, i.e., that most attackers use the result of only a single scanner, is true, as otherwise we would expect to see a much higher number of attacks without a matching scanner. However, of these few non-attributable attacks, more than 60% use more than 24 IPs, the maximum number of amplifiers a single scanner could have possibly found.

For these attacks that use more than 24 IPs, we can further analyze whether they are aggressive scans that exceeded the conservative threshold and are therefore counted as attacks, or whether they are based on the result of multiple scanners. Towards this goal we have to answer the following question: What is the probability of finding x distinct IPs when combining the results of y scans? Intuitively, while it is possible to receive answers from all 48 honeypots with just two scans, it is very unlikely, as this would mean that the second scanner received answers from *exactly* those 24 honeypots that did not answer the first scanner. That is, the likelihood for finding a larger number of x IPs for multiple scanners y increases with y and decreases with x. Formally, this can be modeled as an instance

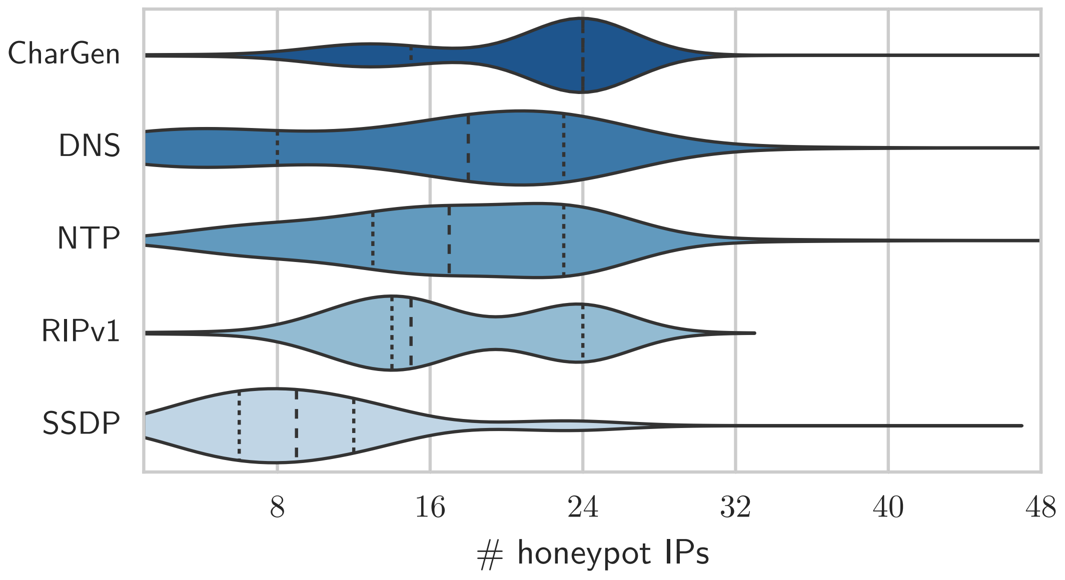


Figure 4.3: Number of abused honeypots per protocol

of the collector's problem with group drawings [142]. In our case, we find that for 80% of the attacks using more than 24 IPs, the corresponding attack sets can be found with a probability of over 60% by combining the results of only two scans. This explains the non-attributable cases in our data set. Rather than being aggressive scans, we conclude that most non-attributable attacks have combined data of multiple scanners.

4.7.3.3 Non-Unique Attacks

Finally, for 237 772 attacks (17.59%), our method found more than one potential scanner, i.e., multiple scanners got replies from the corresponding amplifier set, labeling those attacks as *non-unique*. Intuitively, this can only happen for attacks that use a relatively small amplifier set. Indeed, the average amplifier set size of non-unique attacks is 6.25, i.e., about a fourth of the full response set a scanner could find. In addition, more than 12.88% of the non-unique attacks have abused a single honeypot only.

4.7.4 Improving Attribution Confidence

While we attributed a substantial fraction of all attacks to their scanners with reasonable confidence, for roughly 40% we either found multiple potential scanners or could only attribute the attack to a scan with low confidence. A question that naturally arises is whether this is a inherent limitation of our methodology, or whether it can be alleviated by choosing different parameters, e.g., adjusting the response ratio or leveraging a larger network segment.

To this end, we analyze the influence of the network size and response ratio on the probability that the response set of a scanner is a superset of the attack set. Let N be the size of the

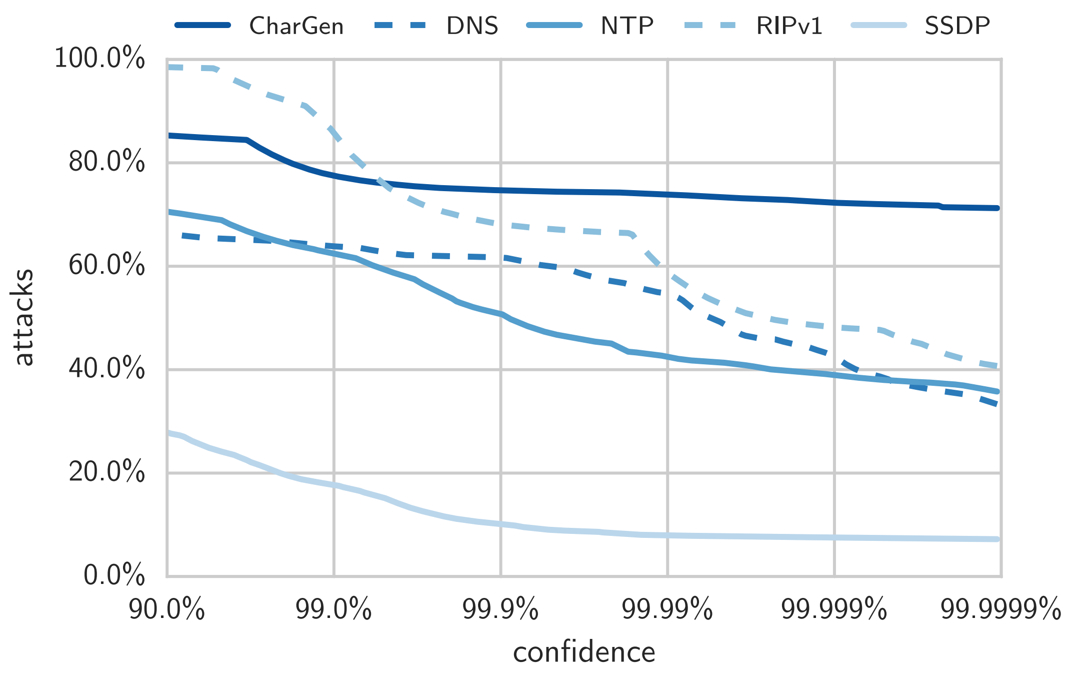


Figure 4.4: Percentage of attacks that could be attributed (y-axis) vs. level of confidence (x-axis)

network, α the response ratio, and A the attack set. Similar to Section 4.7.2, the probability that a scanner received replies from all IPs in the attack set is

$$p = \Pr[A \subseteq R \mid A \subseteq Q] = \frac{\binom{N-|A|}{\alpha N-|A|}}{\binom{N}{\alpha N}}.$$

Since the confidence is computed as $(1-p)^S$, in order to improve the attribution confidence, this probability should be as low as possible.

Interestingly, increasing the network size alone does not reduce this probability:

$$\lim_{N\to\infty} \frac{\binom{N-|A|}{\alpha N-|A|}}{\binom{N}{\alpha N}} = \alpha^{|A|}.$$

Furthermore, this seems to imply that α should be chosen to be very small. However, this is only true if $|A|$ was independent of N and α. Obviously, the choice of α limits the number of IPs a single scanner can find by $|A| \leq \alpha N$ for single scanners.

We therefore analyzed the impact of our network size on the number of chosen IPs in attacks, i.e., the relation between $|A|$ and αN. To this end, we computed the distribution of attack sizes, simulating different network sizes by restricting our dataset to subnets.

We exploited the fact that our data was collected over three /28 networks by performing this analysis three times: using data from just a single subnet (16 IPs, 8 responses), using

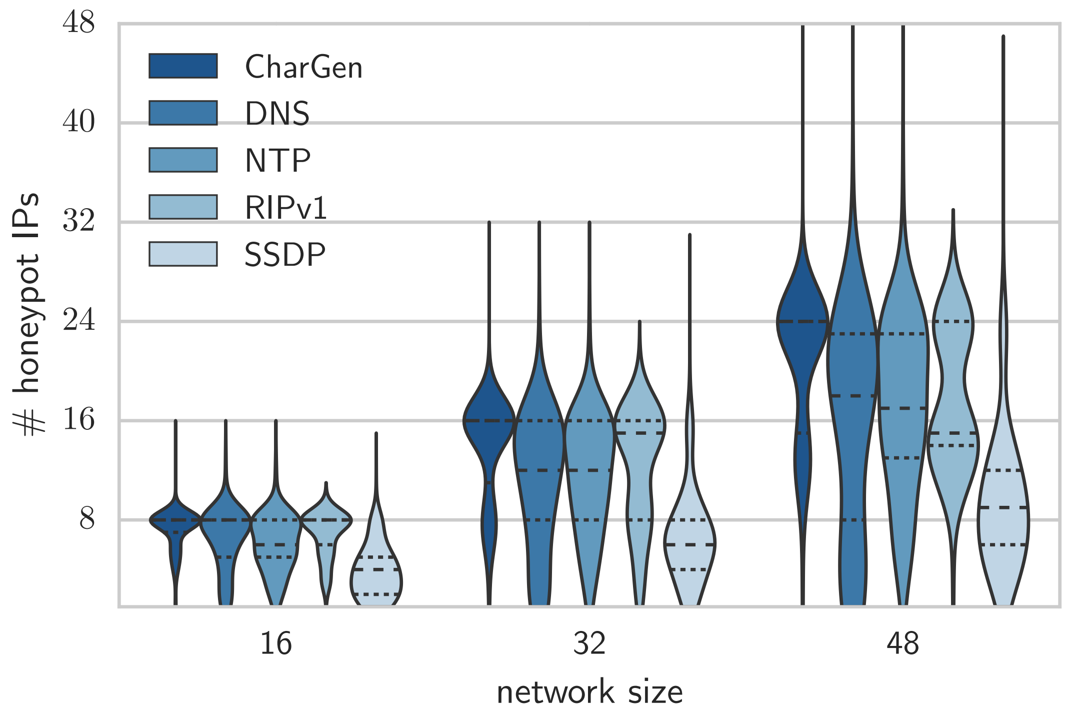


Figure 4.5: Honeypots used vs. network size

data from two subnets (32 IPs, 16 responses), and using data from all three subnets (48 IPs, 24 responses). Restricting the data on a subnet level ensures that the response ratio remains constant, e.g., in the case of 16 IPs, all scanners that scan the entire network will see 8 responses. If we had restricted the data to 16 *random* IPs, some scanners might have received 0 responses, while others might have received 16.

Figure 4.5 shows that the size of the attack sets correlates with the network size. Although our data is too sparse to make strong claims, it suggests that the relation between $|A|$ and αN is linear, with different slopes per protocol.

Assuming a linear relation $|A| = \beta\alpha N$, where β is the protocol-specific slope, we could further investigate the choice of α: We can rewrite the probability from above as

$$\Pr[A \subseteq R \mid A \subseteq Q] = \frac{\binom{(1-\beta\alpha)N}{(1-\beta)\alpha N}}{\binom{N}{\alpha N}} = \frac{((1-\beta\alpha)N)!(\alpha N)!}{((1-\beta)\alpha N)!N!}.$$

Using $\Gamma(n+1) = n!$ we can extend this function to non-integer values, which allows us to find the optimal α numerically for a given N and β using the derivative

$$\frac{\mathrm{d}}{\mathrm{d}\alpha} \ln \frac{\Gamma\big((1-\beta\alpha)N+1\big)\Gamma\big(\alpha N+1\big)}{\Gamma\big((1-\beta)\alpha N+1\big)\Gamma\big(N+1\big)}$$
$$= \big[(-\beta)\psi\big((1-\beta\alpha)N+1\big) + \psi\big(\alpha N+1\big) - (1-\beta)\psi\big((1-\beta)\alpha N+1\big)\big]N.$$

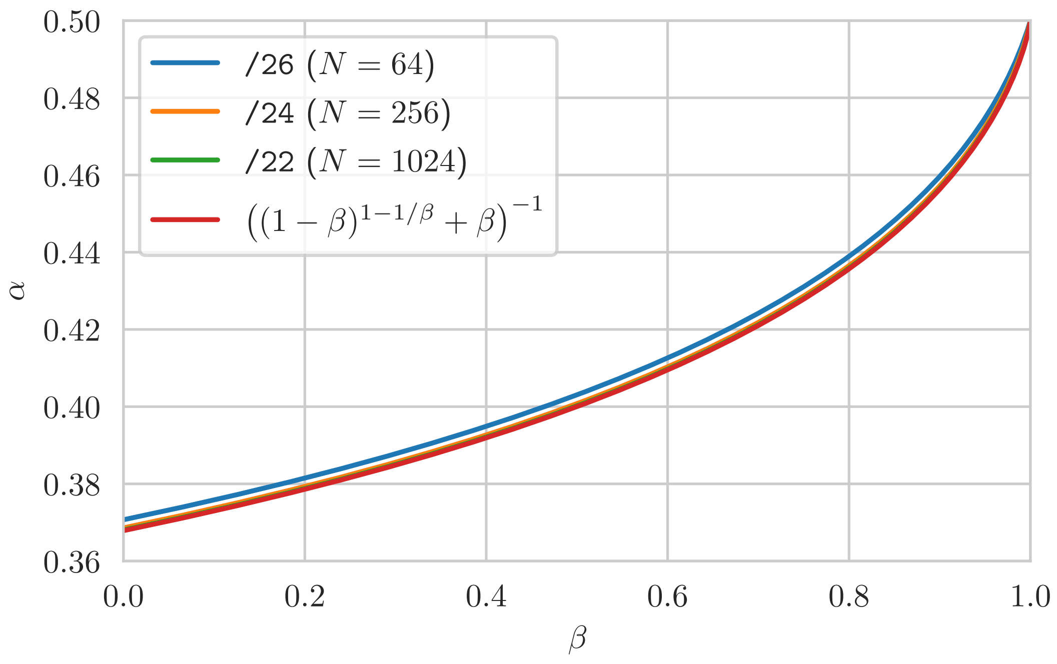


Figure 4.6: Optimal response ratio α for β (numerical results and analytic approximation)

Further, using the crude approximation $\ln(n!) \approx n(\ln(n) - 1) + 1$, we can analytically find that the global minimum lies in the vicinity of

$$\alpha = \left((1-\beta)^{1-1/\beta} + \beta\right)^{-1}.$$

Figure 4.6 shows the optimal value of α for $\beta \in (0, 1)$ computed numerically for three different network sizes, corresponding to $N \in \{64, 256, 1024\}$, as well as the analytical approximation. Counter to intuition, to improve confidence in the case of small attack sets, i.e., small β, one should *also* choose a lower response ratio α. In other words, the gain in confidence by reducing the response ratio α outweighs the gain obtained by increasing the size of attack sets $|A|$ due to α. Furthermore, we find that the above probability is dominated by the term $N!$ in the denominator, and thus increasing the network size N also leads to a dramatic increase in confidence.

4.7.5 A Closer Look at Scanners

After uncovering the scanners providing the reconnaissance behind the attacks, we analyzed the scanners we found in more detail. To this end, we will focus on the 785 285 attacks we could link back to scanners with at least 99.9% confidence. Unless stated otherwise, percentages given in this subsection will be relative to this set of attacks.

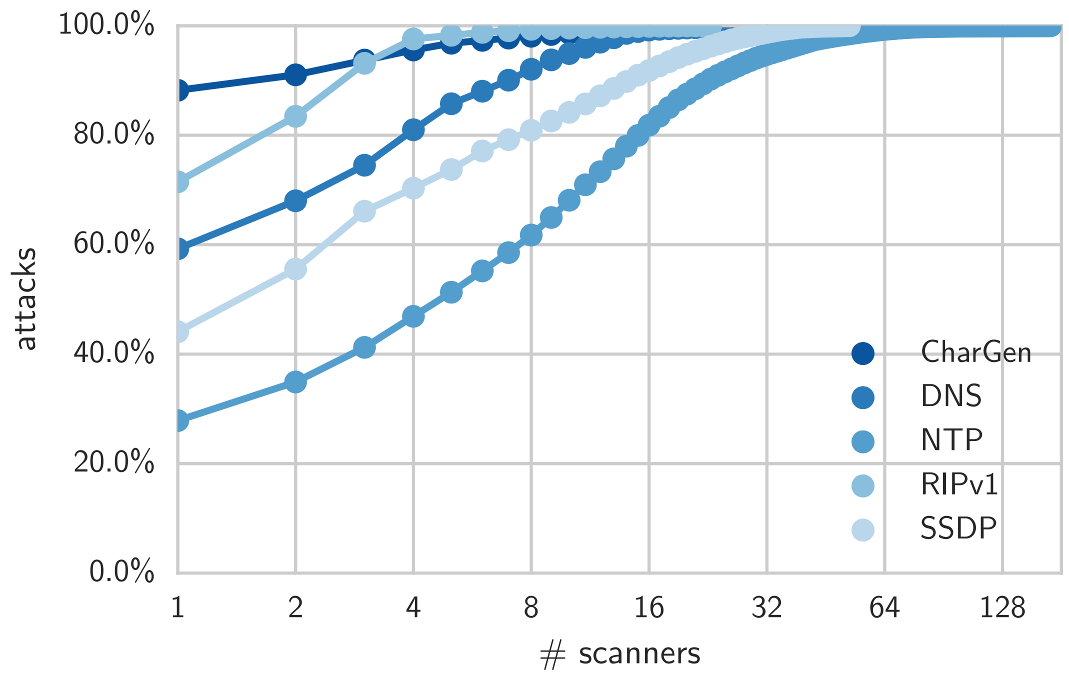


Figure 4.7: Percentage of attributed attacks (y-axis) vs. number of scanners (x-axis, log scale)

4.7.5.1 Attacks vs. Scanners

Interestingly, the 785 285 attacks are based on just 286 different scanners. Furthermore, the number of linked attacks strongly varies per scanner. Figure 4.7 depicts the cumulative distribution function (CDF) over those attacks against scanners. As can be seen, a small number of scanners provided the amplifier sets for the majority of attacks. In the case of `NTP`, 90% of the attacks are based on the scans of less than 20 scanners. For `CharGen`, almost the same fraction of attacks is based only on the amplifier set found by a single scanner. This means that a single scanner provided the data for more than 13% of *all* attacks that our honeypot recorded.

Surprisingly, the cross-protocol share of scanners is quite large, i.e., scanners search for amplifiers of multiple protocols. About a quarter of the scanners (26%) scanned and provided amplifier sets for two or more protocols, in a single case even five protocols.

4.7.5.2 Scanner Locations

To get a better understanding of the virtual and physical locations of scanners, we determined each scanner's autonomous system (AS), as well as the country where the scanner's IP was registered. The geolocation was performed using the freely available `GeoLite2`-database by MaxMind [94]. We determined the autonomous system using the `whois` service run by Team Cymru [67]. If the latter returned no result we conducted a manual lookup by querying the respective regional Internet registry.

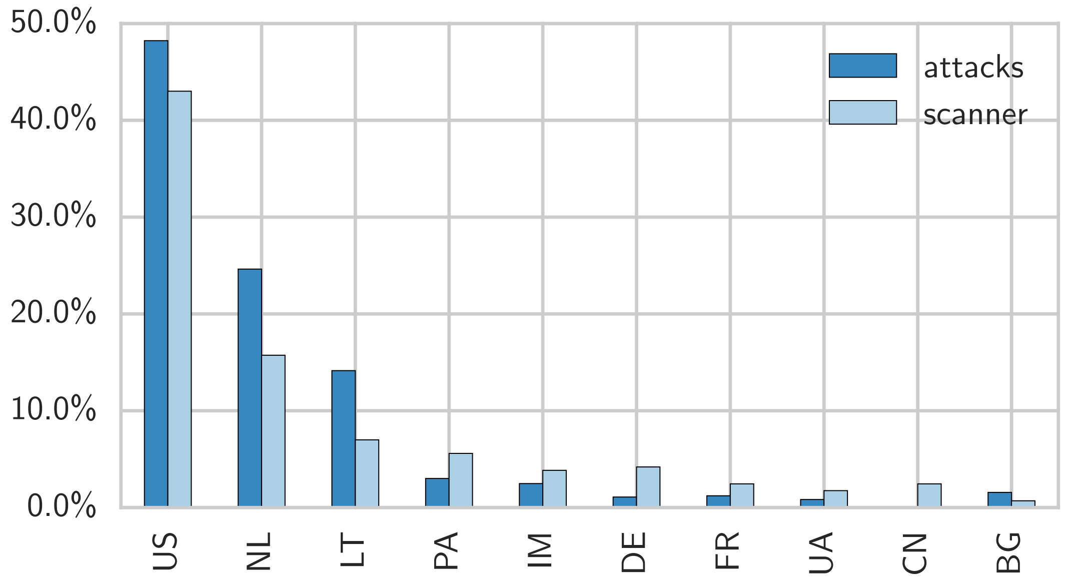


Figure 4.8: Percentage of hosted scanners and attributed attacks per country (top 10)

The 286 scanners we identified are located in 87 autonomous systems, in a long-tail distribution. The most prominent ten AS contain at least 10 scanners each, the top two even at least 25. Overall, the top 10 AS host 156 of the scanners (54.55%). This supports anecdotes that a small number of networks is responsible for large parts of certain abuse types (here: scanning).

Even more surprisingly, the 286 scanners are distributed over only 30 different countries, again in a long-tail distribution. Figure 4.8 shows the percentage of scanners located in and the percentage of attacks attributed to scanners in the top 10 countries. 3/4 of all scanners are hosted in the US, the Netherlands, Lithuania, Panama, or Germany, with the vast majority of them being located in the US. Interestingly, scanners from the US, the Netherlands, and Lithuania have an above-average number of attacks attributed to them. In fact, over 87% of all attacks were attributed to scanners in those three countries.

4.8 Mapping Scan Infrastructures to Attack Infrastructures

Mapping scans to attacks already allowed us to find one important part of the adversarial infrastructures, namely those systems that perform Internet-wide scans to prepare the attacks. In this section, we turn to the infrastructures that are used to *perform* the attacks. The main question that we would like to answer is the following: Are the systems used to perform scans also used to perform subsequent attacks? In fact, the technical requirements to launch attacks are very similar to those needed for Internet-scale scans. Both parts require a powerful network connection, and combining the infrastructure would certainly make attacks easier, as there is no need to exchange information. In the following, we therefore answer whether attackers reuse their scan infrastructure for launching attacks.

4.8.1 Methodology

In a DDoS amplification attack, the attacker sends out requests to a set of amplifiers, but spoofs the packet header to inject the victim's IP address. Therefore, from the amplifier's perspective, packets observed during an attack will only contain the IP address of the victim. Worse, the victim will only see traffic originating from the amplifiers. Finding the actual packet source of amplification attack is thus a non-trivial problem—irrespective of the perspective.

To tackle this problem, we propose to combine our honeypot data with trilateration techniques to trace back the packet origin. To this end, we leverage the time-to-live (TTL) field in the packet header. When sending out packets, the sender chooses an initially high TTL value, and every hop along the route to the destination will decrease the TTL value by one. Thus, the difference between the initial TTL and the received TTL can be used to estimate the length of the route between the sender and the receiver. Having multiple globally-distributed vantage points to take TTL-based measurements between the source and the honeypots allows comparing two locations on the network following the concept of trilateration. In other words, if the locations of honeypots are wisely distributed, and if attacks abuse many honeypots at the same time, the honeypots allow measuring the path lengths between the packet origin and the various honeypot placements. This will help to approximate the packet origin: Our hypothesis is that packets from the same source will have equal (or at least similar) hop distances between the origin and our honeypots and that the initial TTL set by the sender is fixed. We back up this hypothesis with the observation that in 92% of all attacks no honeypot observed more than 3 distinct TTL values. If we thus compare the hop distances recorded at the honeypots, we can say if two packets originate from the same system by finding similar TTL distances—without relying on the IP addresses.

Formally, the TTL recorded at a receiver r is equal to the initial TTL set by the sender s minus the hop count distance $d_{s,r}$, i.e., $ttl_r^s = ttl_s - d_{r,s}$. Assuming that the sender uses a fixed initial TTL value[1], the location of a source s relative to a set of n receivers $r_1, \ldots, r_n$ can then be modeled as an n-dimensional vector capturing the distances d_{s,r_i} between the source and the receivers:

$$\vec{d}_{s,\vec{r}} = ttl_s \cdot \vec{1} - \vec{ttl}_{\vec{r}}^{s}$$

$$\begin{pmatrix} d_{s,r_1} \\ d_{s,r_2} \\ \vdots \\ d_{s,r_n} \end{pmatrix} = ttl_s \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} - \begin{pmatrix} ttl_{r_1}^{s} \\ ttl_{r_2}^{s} \\ \vdots \\ ttl_{r_n}^{s} \end{pmatrix}$$

To compare the location of two sources, we will use the $\ell 1$ distance (also known as the Manhattan distance or rectilinear distance). However, this is not trivial, as our model has so far assumed that the initial TTL set by the sender is known. While our data seems to suggest

[1]This is usually the case as we will discuss in Section 4.9.2.

that most attacks indeed choose the maximum of 255 as the initial TTL value, attackers may also choose different values. Consequently, we do not assume a specific initial TTL value. In order to circumvent the missing initial TTL, we decided to "align" measurements as follows: The $\ell 1$-distance of two sources s_1 and s_2 is computed as

$$\left\|\vec{d}_{s_1,\vec{r}} - \vec{d}_{s_2,\vec{r}}\right\|_1 = \left\|\vec{ttl}_{\vec{r}}^{s_2} + \vec{ttl}_{\vec{r}}^{s_1} + \left(ttl_{s_1} - ttl_{s_2}\right) \cdot \vec{1}\right\|_1$$

and depends only on the *difference* between the initial TTL values. To "align" two measurements $\vec{ttl}_{\vec{r}}^{s_2}$, $\vec{ttl}_{\vec{r}}^{s_1}$, we find t (a "shift" value in the range $[-255, 255]$) that minimizes the following distance between two TTL vectors:

$$\left\|\vec{ttl}_{\vec{r}}^{s_2} + \vec{ttl}_{\vec{r}}^{s_1} + t \cdot \vec{1}\right\|_1 .$$

Intuitively, the more measurement points (i.e., honeypot locations) we have, the more accurately we can compute the TTL-wise distance between two sources. However, while our modified honeypot listens on 48 IP addresses, all those IP addresses point to the same system and therefore likely have identical routes. Obviously, a single measurement point is not sufficient to perform true trilateration. Fortunately, we can leverage the 11 unmodified AmpPot instances that are deployed in various locations and therefore should observe different routes. Combining their dataset with our Selective Response honeypot provides us with up to 12 measurement points, yielding an entropy that is sufficiently high to perform trilateration, as we will show in the following.

4.8.2 RIPE Atlas Probes

The TTL distance should approximate whether packets stem from the same source, in that "small" distances hint at similar packet sources. However, given Internet route changes and load balancing, it is unclear what distance we need to tolerate to spot same-origin packets. To validate whether our TTL metric is indeed meaningful and does not create false positives, we use a ground truth dataset. To this end, we leverage the Atlas project by RIPE [117]. In Atlas, volunteers host *probes*, small devices used to carry out measurements on the Internet such as "ping" or "traceroute". Measurements can be performed by anyone in exchange for a certain amount of credits, which can in turn be earned by hosting probes.

To establish our ground truth, we selected a random set of 200 probes and instructed them to send packets to our honeypots. At the honeypots we then recorded the TTL values of the traffic coming from these probes. We were interested in the stability of routes at two time scales. To measure changes in the hop count in the order of minutes, we sent three packets at intervals of two minutes. To measure changes in the order of hours, we repeated this process five times at intervals of six hours. After excluding probes that only sent partial data (or no data at all), our dataset contained TTL values for 168 distinct sources for five measurements.

Our random selection of probes guaranteed creating a heterogeneous set in terms of probe locations. That is, we had probes with a large distance from each other, as well as clusters of probes from a small dense region, such as the Amsterdam area in the Netherlands. This

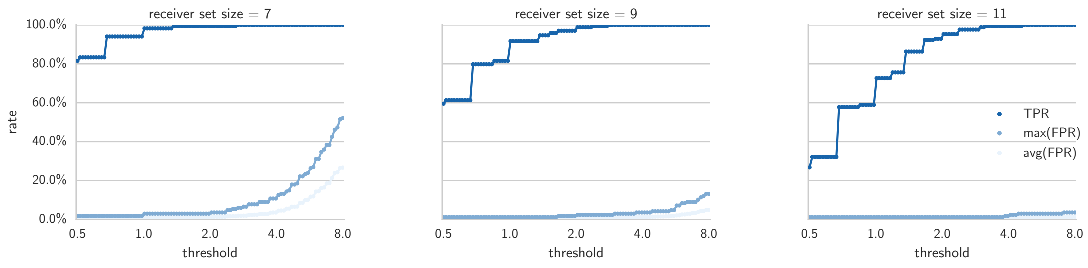


Figure 4.9: TPR, maximum FPR, and average FPR for receiver-set sizes 7, 9, and 11

was done to confirm the intuition that distant sources have very different routes, and to investigate whether different sources in the same proximity would be mistaken for one another—assuming that they share a large amount of routes.

To measure if our trilateration methodology would mistakenly flag two different sources as being the same, we computed the minimal $\ell 1$-distance between every pair of sources. Additionally, we also investigated the influence of the number of receivers on the resulting distance. Intuitively, given that distances sum up, a higher number of receivers could lead to a higher $\ell 1$ distance. To this end, we sampled random honeypot subsets of size $2, 3, \ldots, 11$ for each pair of sources and computed the minimal $\ell 1$ distance between the pair using the TTL values recorded by this subset.

From these distances we could then derive thresholds such that measurements with a distance below the threshold are likely to stem from the same source, while measurements with a distance above the threshold are more likely to stem from different sources. In order to measure the performance of a given threshold, we turned to two well-known measures from classification, namely the *true positive rate* (TPR), measuring the fraction of sources that could be correctly re-identified, and the *false positive rate* (FPR), measuring the fraction of sources falsely assumed to be identical. More formally, a *true positive* means that a probe had two measurements with a distance below the threshold, while a *false positive* corresponds to two measurements from two *different* probes that had a distance below the threshold. However, since every probe can be confused with every other probe, a global FPR is not applicable. Instead, we compute the FPR *per probe*.

We give example curves of the TPR, the average FPR, and the maximum FPR in Figure 4.9 for 7, 9, and 11 receivers, respectively. As expected, a smaller number of receivers increases the FPR. For example, using a threshold of 8 leads to a FPR of over 50% in the worst case when using just 7 receivers. Increasing the number of receivers to 11 decreases the FPR to below 5%. Furthermore, smaller thresholds decrease the FPR, but also lead to a loss of TPs.

This leads to the question of how the threshold should be chosen. Since we are mainly interested in learning if a scanner infrastructure is also used to launch attacks, we focus on the FPR. In a similar fashion to Section 4.7.2, we can fix a *confidence level* and derive a threshold for a given number of receivers: Since the FPR estimates the probability with which our method gives false accusations, the complementary probability of this corresponds to the level of confidence, i.e., the probability that the attribution is correct. Figure 4.10

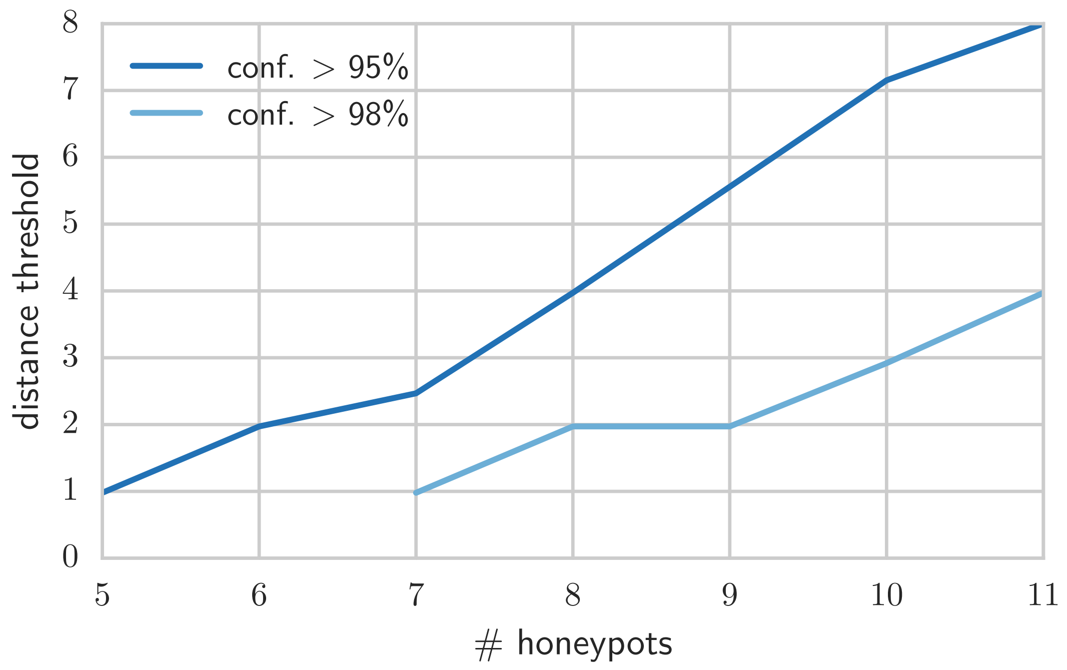


Figure 4.10: TTL distance thresholds (y-axis) depending on the receiver set sizes (x-axis)

states the thresholds per receiver set size for a confidence level of 95% and 98%, respectively. For example, two events observed by a set of 8 honeypots stem from the same source with 95% confidence if their TTL distance is below 4; to have a confidence of 98% their distance should be below 2.

4.8.3 Malicious Scanners

Knowing which thresholds to choose, we will now apply our methodology to our dataset of scanners. That is, by comparing the TTL vectors of a scan event and an attack, we now answer the question whether the infrastructure used to perform the scans is also used to launch attacks.

During an attack, packets are typically sent quasi-simultaneously to the honeypots. This is not necessarily true during scans, as a scanner may distribute its activities over a longer period of time. Therefore, to compare the TTL values between scanners and their attributed attacks, we computed the distance between the TTL values observed in the attack and the *chronologically closest* scan event for each honeypot. This minimizes the effects of potential route changes. To account for the fact that we might see small fluctuations of TTL values during an attack, we compare the scan against the average TTL value measured at a honeypot. Moreover, to stay consistent with the parameters of our Atlas-based measurements, we also limited the analysis to attacks that took place within 24 hours before or after a scan. Although this might sound like a strict limit, it excludes only six out of the 286 scanners (2%) we identified.

Under the assumption that the true distribution of scan and attack infrastructures is somewhat similar to the distribution of our Atlas probes[2], the thresholds found in the previous section are still applicable here. Using those thresholds we find that with confidence of 95% or more, 44 of the 286 scanners are presumably also launching attacks, 34 of them even with a confidence of 98% or higher. Furthermore, since these 34 scanners have been found responsible for 293 478 of the attacks with a confidence of 99.9% or higher, we conclude that our methodology successfully uncovered the true attack infrastructure behind over a third of the attacks for which we could find a scanner with 99.9% confidence, which equates to a fifth of *all* attacks observed at our honeypots.

4.9 Discussion

Our novel methodology helps to identify infrastructures of current state-of-the-art attacks, which is significant progress in terms of finding the origin of amplification attacks. Having said this, we will now discuss some of the assumptions of our methodology and discuss how adversaries might be able to evade our attribution process in the future.

4.9.1 Single Scanner

As a primary assumption, we assumed that the amplifier set used in DDoS amplification attacks is scanned from a *single* public IP. This seemingly strong assumption is backed by two arguments. First, our results show a very small fraction (2.52%) of attacks for which no potential scanner could be found. Were attackers to compile their amplifier sets from multiple sources, we would expect a much higher number of attacks marked as *non-attributable* (see Section 4.7.3).

Secondly, attackers need to rescan for amplifiers at regular intervals due to amplifier IP address churn. Kührer et al. showed that typically less than half of the amplifiers are still reachable a week after the scan [80]. Periodic scans require a setup capable of scanning the entire IPv4 address space and suitable for long-term scan operation. Since launching large-scale scans violates most hosting providers' terms of service, we argue that maintaining such scanners incurs a non-negligible amount of work. Furthermore, when performing an Internet-wide scan, one would not expect to achieve a much different result when scanning from another source. All this combined leads us to the conclusion that attackers are not incentivized to maintain multiple scanners.

Having said this, combining the results of multiple scanners could evade our attribution in its current form. To tackle this problem, one could increase the network size N and reduce the response ratio α, such that our selective response scheme guarantees even higher entropy and also combinations of two or more scanners could be re-identified.

[2]The Atlas probes were chosen at random from a globally-distributed set of systems. Lacking any ground truth on true attack sources, we had no sound methodology to further verify or invalidate this assumption.

4.9.2 Initial TTL

When comparing scanner infrastructure to attack infrastructure, we assumed that the initial TTL set by the scanner and/or attacker was constant for all packets. This holds true for packets carrying non-spoofed headers, as the network stack of common operating systems will typically use a default value (usually one of 64/128/255, depending on the operating system). But even in the case of attack traffic we see fixed TTLs for the majority of attacks, in accordance with the observations made in our earlier studies [S1].

Unfortunately, attackers could evade our infrastructure comparison by randomizing their initial TTL values. However, we may be able to average the various TTL values to tolerate such randomizations, as the average survives randomization. While randomization is thus not an effective evasion technique, there are smarter ways our TTL-based methodology can be fooled, such as randomly choosing a single initial TTL *per amplifier*.

Worse, an attacker may try to provoke a *false* attribution result by choosing its initial TTL values to match those of the corresponding scanner. While it has been shown that such attacks are possible [10], we believe that no attackers currently employ such complex deception mechanisms in practice.

4.9.3 Honeypot-Driven Observations

Our selective response methodology assumes that attackers leverage sufficiently many amplifiers in an attack to allow for the attribution process. Related studies have shown that literally thousands of amplifiers are involved in attacks [124], supporting this assumption. However, an attacker may select amplifiers such that only a few honeypots see any attack traffic. For example, in our current deployment, an attacker could limit the number of amplifiers per subnet. Note that our methodology does not actually require that honeypots are located in the same subnet. The same scheme could be applied to honeypots scattered among various providers and in disjoint subnets, mitigating this problem.

Finally, attackers could aim to identify honeypots running AmpPot by its behavior and avoid their use. To avoid probe response attacks [15], access to AmpPot data is restricted to vetted entities, such as researchers and law enforcement agencies. Still, in its current operational mode, AmpPot can be identified, as it emulates certain protocols, usually even with a fixed response. However, as a countermeasure, one could configure AmpPot to run in a “proxy mode” that intermediates traffic between actual service implementations with amplification vulnerabilities (such as NTP servers). This would make the honeypot indistinguishable from other amplifiers. Note that such a stealth mode may also have implications on the rate-limiting functionality of honeypots, as limiting the traffic may potentially be another way to identify the honeypot.

4.10 Related Work

Closest to the work presented in this chapter, our original publication on AmpPot [S1] also describes early attempts to attribute the scanners that helped to prepare the attacks. However, there the attribution is limited to the scan *software* (e.g., Zmap [44] or masscan), while here our goal is to track the scan *origin*. Furthermore, we have significantly extended the AmpPot design with a novel idea of a probabilistic response scheme, which is a fundamental enhancement to foster tracebacks.

4.11 Conclusion

Our novel methodology links scan infrastructures to amplification attacks, which is a major breakthrough in the process to identify the origin of amplification attacks, one of the major threats on the Internet. We found cases where the scan infrastructures are also used for attacks, i.e., we could even pinpoint the attacker down to the infrastructure that she used to perform the attacks. We have shared our findings with law enforcement agencies (in particular, Europol and the FBI) and a closed circle of tier-1 network providers that use our insights on an operational basis. Our output can be used as forensic evidence both in legal complaints and in ways to add social pressure against spoofing sources. Such pressure has led to successful interruptions of other malicious activities in the past, such as the significant drop of spam volume after the McColo shutdown in 2008 [68, 29].

Admittedly, attackers may attempt to evade some parts of our methodology to fly under the radar. We acknowledge the possibilities for evasion, but still believe that our work is of great help to resolve the current situation of amplification sources. In the long run, we will have to seek more conceptual solutions to the problem, such as an Internet architectures that guarantee address authenticity by design (and thus prevent spoofing) [169], schemes that guarantee bandwidth reservations regardless of DDoS attacks [13], or efforts to close amplification vectors in Internet protocols [80, 33].

5
Linking Amplification DDoS Attacks to Booter Services

5.1 Motivation

Although our scanner attribution presented in the previous chapter constitutes a major step towards attack traceback, in many cases it only allows us to find systems used to prepare for attacks, but not the actual attacking systems. To further our efforts in that direction, in this chapter we focus on a subclass of attacks, namely those launched by DDoS-for-hire systems, so-called booter services. Specifically, we present techniques for attributing amplification DDoS attacks to booter services that launched the attacks. Our k-Nearest Neighbor (k-NN) classification algorithm is based on features that are characteristic for a DDoS service, such as their choice of reflectors or their network location. This allows us to attribute DDoS attacks based on observations from honeypot amplifiers, augmented with training data from ground truth attack-to-services mappings we generated by subscribing to DDoS services and attacking ourselves in a controlled environment. Our evaluation shows that we can attribute DNS and NTP attacks observed by the honeypots with a precision of over 99% while still achieving recall of over 69% in the most challenging real-time attribution scenario. Executing our k-NN classifier over all attacks observed by the honeypots shows that 25.53% (49 297) of the DNS attacks can be attributed to 7 booter services and 13.34% (38 520) of the NTP attacks can be attributed to 15 booter services. This demonstrates the potential benefits of DDoS attribution to identify harmful DDoS services and victims of these services.

5.2 Problem Description

Distributed Denial-of-Service (DDoS) attacks have become commoditized by DDoS-for-hire services, commonly called *booters* or *stressers* [72, 123]. A large number of booter services advertise their services openly as an economical platform for customers to launch DDoS attacks. At the same time DDoS attacks are increasing in number and in magnitude. This proliferation of DDoS attacks has caused many network and website operators to rank this type of attack as one of the largest threats facing them [6]. This barrage of DDoS attacks has increased the demand for Content Delivery Networks (CDNs) and Software Defined Networking defenses that can absorb and filter these attacks [58]. In turn, this has prompted attackers to react by devising increasingly efficient methods of bypassing or overwhelming defenses. The result is an escalating technological arms-race between DDoS attackers and defenders that at times has congested segments of the core Internet infrastructure as collateral damage [112].

Despite the proliferation of DDoS services and attacks, little progress has been made on attributing the services that are launching these attacks on behalf of their customers. Most ideas for attribution focus on IP traceback mechanisms [108, 164, 140, 137, 127] to trace the source of spoofed IP packets, which require ISPs to assist and so far have not been widely deployed. This has resulted in most of these attacks being unattributed unless the attackers unveil themselves. While it is important to create strong technological DDoS defenses, we argue that there is also benefit in investigating other methods that enable attribution of DDoS attacks to the services responsible for launching these attacks. For instance, some of these booter services—seven out of 23 services that we studied—claim they are benign services by advertising as “stress-testing” services intended to be used only by authorized administrators. For example, one of these services included this statement on their website, *“We provide a professional and legal ip stresser service which is based on a massive 20 dedicated server backend, ensuring that your server is tested to its limits.”* Attribution can remove this veil of legitimacy and assist efforts to undermine these services by allowing victims and law enforcement to attribute which booter services were responsible for an attack. Attribution also enables measuring the scale of these services and prioritizing undermining the larger services that are causing more harm. In order to assist ongoing investigations, we are continually sharing information from our study on DDoS attacks and booter services with the European Police Office (Europol), the United States Federal Bureau of Investigation (FBI) and large ISPs or backbone providers.

5.3 Contributions

In this chapter, we show that it is possible to build supervised learning techniques that allow honeypot amplifier operators to accurately attribute attacks to the services that launched them. To begin, we identify three key features that honeypot operators can record to construct a supervised k-NN classifier that can attribute attacks. In order to validate our method, we subscribed to 23 booter services and generated a ground truth data set of attacks to booter service mappings[1]. Validation of our classifier using the ground truth self-attack data set shows that it is highly precise at attributing DNS and NTP attacks with a precision of over 99% at 69.35% recall in the worst case of real-time attribution. When retrospectively attributing attacks, the recall even increases to 86.25%. Executing our classifier over the set of all attacks observed by the honeypots shows that 25.53% (49 297) of the DNS attacks can be attributed to 7 booter services and 13.34% (38 520) of the NTP attacks can be attributed to 15 booter services.

Our findings demonstrate that many of the attacks we observed can be attributed to a small set of booter services that are operating relatively openly. Our ability to attribute large numbers of attacks to a small set of booter services and sharing of this information with Europol and the FBI to assist in active investigations demonstrates the usefulness of our attribution methods.

In summary, we frame our contributions as follows:

- We present a k-NN-based classifier that attributes amplification DDoS attacks observed by honeypots with a precision of over 99% while still achieving recall of over 69% in the most challenging real-time attribution scenario.
- We attribute 25.53% (49 297) of the DNS attacks to 7 booter services and 13.34% (38 520) of the NTP attacks to 15 booter services.

[1] Our ethical framework for these measurements is based on previous studies that have used this methodology [72, 124].

5.4 Background

5.4.1 Threat Model

In this chapter, we are concerned with a special type of attacker: *booter services*. These offer platforms for DDoS-as-a-service, often under the disguise of "stress-testing", where customers can request various types of attacks for a small fee. The booter will then launch these attacks using its infrastructure. Our threat model thus contemplates four parties: *Customers*, who commission attacks; *booters*, who conduct the actual attacks; *amplifiers*, who are exploited to amplify traffic; and *victims*, who are the targets of such attacks.

The aim of this chapter is to attribute attacks observed at honeypot amplifiers to booter services. This is non-trivial, as from an amplifier's perspective, the requests seem to be legitimate requests by the victim (due to the use of spoofed source IP addresses by the booter). While ultimately one would like to also identify the customer who commissioned the attack, only the booter, amplifiers, and the victim are directly participating in an attack. Nonetheless, since the booter has a business relation to the customer, pinpointing the booter behind an attack constitutes an important step towards this goal.

5.4.2 Ethical Considerations

As part of our study we subscribed to 23 booters and conducted a controlled set of self DDoS attacks. Furthermore, we also leveraged honeypots for amplification attacks. We settled on this methodology for collecting a ground truth data set of mappings between observed attacks and the services that launched these attacks after finding that no data set available to us could be used to validate our DDoS attribution techniques. Before we began performing these self DDoS attacks we carefully attempted to minimize the harms and maximize the benefits associated with our methodology based on observations from previous studies that launch self-attacks in order to measure booter's attacks [72, 124].

We received an exemption from our Institutional Review Board (IRB), since our study did not include any personally identifiable information. In addition, we consulted with our institution's general counsel, who advised us not to engage with any DDoS service that advertised using botnets and to cease active engagement with any booter service that we realized was using botnets.

An analysis of TTL values observed by the honeypots indicated that it is unlikely any of the booter services we subscribed to used botnets. Based on the guidance of our institution's general counsel, our victim server was connected by a dedicated 1 Gbit/s network connection that was not shared with any other servers. We also obtained consent from our ISP and their upstream peering points before conducting any DDoS attack experiments. Finally, we minimized the attack durations, notified our ISP before launching any attack and had a protocol in place to end an attack early if it caused a disruption at our ISP.

We purchased subscriptions from 23 booter services. To minimize the amount of money given to these services, we selected the cheapest option, which ranged from \$6-\$20 and averaged \$12 per month. In total, we spent less than \$400 and no individual booter service received more than $40 in payments as part of the measurements in this chapter[2]. All payments were made using PayPal and we assumed that proper controls were put in place at PayPal to mitigate the risk of money flowing to extremist groups. As part of our design methodology, we minimized the amount of money paid and targeted a small set of booters to obtain a valuable ground truth data set.

Our method created some harm to amplifiers and their upstream peering points by consuming bandwidth resources. The largest amount of bandwidth consumed was 984.5 kbit/s for NTP amplifiers and the least was 16.7 kbit/s for DNS amplifiers, similar to those reported in a previous study [72].

Over the course of our experiments we did not receive any complaints from the operators of these amplifiers. We limited our attacks to 30 seconds. Based on analysis from a previous study that used a similar methodology [72], these short duration attacks enable us to observe about 80% of the amplifiers used by a given booter service and reduce the harm we cause to misconfigured amplifiers.

Similarly, the use of DDoS honeypots might also incur harm on the Internet. We again used AmpPot [S1] to observe attacks. As detailed in Section 2.2, AmpPot limits the rate of requests to avoid contributing to DDoS attacks and deploys automatic IP blocking: The honeypots will stop responding for one hour to any IP address sending more than 10 requests per minute. This limits the maximum amount of data sent to a DDoS victim to a few kilobytes.

5.5 Amplification Attack Data Set

To investigate if and how amplification attacks can be attributed to their originating booter service, we established two data sets: The first dataset (Section 5.5.1) consists of amplification attacks recorded by AmpPot and provides us with a global, yet incomplete view of attacks, since only the victim addresses can be recorded. The second dataset (Section 5.5.2) therefore consists of attacks which we launched against ourselves using booters in a controlled fashion to learn about attack techniques and characteristics of certain attackers.

5.5.1 Honeypot Attacks

To collect insights into the overall landscape of amplification attacks, we leverage data collected by AmpPot [S1]. As shown in Section 2.2, AmpPot emulates several UDP-based protocols that have known amplification vectors and will thus eventually be abused as part of real-world DDoS amplification attacks. Attackers will eventually find such honeypots via Internet scans, and start abusing them as potential reflectors shortly thereafter. AmpPot thus serves as an eye on global amplification attacks, and due to the nature of the attack traffic, can observe who is being attacked and when.

For this chapter, we use data collected over two months, from December 9, 2015 to February 10, 2016. During this time, AmpPot deployment consisted of eleven globally-distributed

[2]To put this into perspective: Previous studies of these booters have shown that they have thousands of paid subscribers and generate revenues of over $10 000 per month [72, 123].

	AUR	BAN	BO1	BO2	BO3	CRI	DOW	EXI	EXO	KST	INB	NET	RAW	SER	STA	ST1	ST2	ST3	ST4	SYN	THU	VDO	WEB
CharGen	✓	✓	✓	✓	✓		✓		✓	✓		✓	✓	✓					✓	✓	✓		✓
DNS	✓	✓		✓	✓		✓	✓			✓		✓	✓	✓	✓	✓	✓		✓	✓	✓	✓
NTP	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
SSDP	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓			✓	✓	✓		✓		✓	✓	

Table 5.1: Covered booter services

honeypots with single static IP addresses as well as a twelfth honeypot employing *Selective Response* on an additional 24 IP addresses (see Section 4.6). In this period, the honeypots observed a total of 570 738 amplification attacks (8 918 attacks per day on average). Given that the vast majority of these (95%) use one of CharGen, DNS, NTP, or SSDP, we only focus on those four protocols for our further analyses.

5.5.2 Self-Attacks

The honeypots give us valuable insights into global attacks, but do not give us indications where the attacks were coming from. Previous studies have identified so-called booter services ("booters") as being responsible for a large number of amplification attacks [71, 123, 124, 72]. In an attempt to learn attack characteristics of these booters, we signed up at a number of these services and then launched short-lived amplification attacks against a target under our control.

To start launching self-attacks and correlating them with the traffic seen at the honeypots, our first task was to identify booter services to cover in the study. Absent a centralized location for finding booters, we located services via search engines and advertisements on underground forums. We selected a total of 23 services offering amplification attacks based on NTP, DNS, CharGen and SSDP. When selecting these booters, we tried to include services that we speculated to be more stable and have more subscribers based on reviewing user feedback on underground forums. To minimize the amount of money we paid to these abusive services, we kept the number of covered booters relatively small.

Table 5.1 provides an overview of the booter services[3] that we cover and the amplification attack types they offer. NTP was the most popular attack protocol, followed by DNS. 16 of the 23 services clearly advertise malicious DDoS attacks. In contrast, seven services hide their malicious intention behind "stressing" services, a seemingly benign way to verify the resilience of a network against DDoS attacks. However, not a single service performs any kind of attack target validation. That is, service subscribers can specify any IP address (or domain) that should be attacked, regardless of whether the target is under the control of the client. This shows the clear malicious intention behind all 23 booter services.

Booter services maintain front-end sites that allow their customers to purchase subscriptions and launch attacks using simple web forms. We created custom crawlers to automate the

[3]To avoid unintentionally advertising booter services covered in this study, we replace the name of booter services by the first three letters of their domain name. The last letter is replaced by a number in the case of name collisions.

		attacks	observed		
Protocol	**booters**	**launched**	**at victim**	**at honeypots**	**>100 pkts.**
CharGen	16	608	417	35	33
DNS	19	676	452	173	100
NTP	22	823	577	421	373
SSDP	16	560	351	1	0
Total	23	2 667	1 797	630	506

Table 5.2: Overview over self-attacks

task of visiting the websites of covered booters and launching attacks directed at our own target. Using this automation, daily attacks were launched for each covered booter and attack type. A total of 13 booter services were covered within the first week of starting the self-attacks on December 9, 2015 and by January 14, 2016 all 23 booters were covered.

Labeling Self-Attacks As we instructed all booters to attack the same target, we had to find a mechanism to separate between multiple consecutive self-attacks to assign (booter) labels to the attack traffic. To this end, we initially relied on the time that attacks were initiated. To account for clock skew, we left 10 minutes between consecutive attacks and used a grace period of ±3 minutes for matching. On January 14, we started to use a distinct victim IP per booter service as an improved matching criterion. Based on the same criterion, we then also mapped the self-attacks to attacks observed at the honeypots.

Table 5.2 gives an overview over the self-attacks. We launched a total of 2 667 CharGen, DNS, NTP and SSDP attacks using 23 booter services. Interestingly, only around 2/3 of the attacks we initiated were observed at the victim. This can be explained by our observation of maintenance issues that some booter websites have. Sometimes booter websites provide the user interface for selecting a particular attack type that is temporarily non-functional. To users it appears that the attack has been successfully launched, but no actual attack traffic is generated as a result of initiating such attacks.

Of our self-attacks, the honeypots observed many NTP (73.0%) and DNS attacks (38.3%), but only a small fraction of the CharGen attacks (8.4%) and just a single SSDP attack. Furthermore, while *some* traffic could be observed for 630 attacks, in only 506 cases did we record more than 100 requests. To understand why the honeypots missed large portions of SSDP and CharGen attacks, we investigated the attack traffic towards our victim to learn the preferences of attacks in choosing reflectors. In both cases, we found that the vast majority of the reflectors that were abused by multiple booters send responses that are significantly larger than the ones configured in AmpPot. This indicates that the honeypots' SSDP and CharGen responses were too small to be attractive for attackers, and adversaries preferred other reflectors with better amplification. We leave further investigations on reflector selection strategies open for future work and focus on DNS and NTP in the following.

Multi-Branding Booters During the sign-up phase, we noticed that some booters were visually similar. Investigations have revealed that one miscreant follows a multi-branding strategy, i.e., sells the same service via different booter names that shared similar web front-ends. It became apparent that attacks from `RAW` and `WEB` shared characteristics, and also their sign-up page of the web interface was equivalent in appearance and HTML code. We further analyzed those two booters by launching application layer (layer 7) attacks against our victim server. Layer 7 attacks usually abuse public HTTP proxy servers to hide the identity of back-end servers involved. However, some proxies reveal the identity of the requesting clients in the `X-Forwarded-For` field of the HTTP header. Based on this observation, we were able to verify that these two booters used shared back-end infrastructure. We thus conclude that `RAW` and `WEB` are likely to be operated by the same individuals and will regard them as equivalent.

5.6 Characteristic Attack Features

We will now introduce characteristic attack patterns that we can use to train our classifier for attribution purposes. We first describe various characteristics that we have observed to repeat across subsets of attacks at the honeypots. We then describe how we leverage these observations as features to summarize attacks.

5.6.1 Attack Observations

While analyzing the attacks captured by the honeypots, we observed the following three properties that repeated across subsets of the attacks.

Honeypot Sets: Although eleven honeypots were active since the end of 2014, few attacks (1.63%) abused all of them simultaneously. In fact, more than 60% of all DNS- and NTP-based attacks abused five honeypots or less. This indicates that attackers either perform only partial scans of the Internet, or choose a subset of the discovered amplifiers in subsequent attacks.

Interestingly, we observed that honeypot sets seem to be *reused* across multiple attacks, i.e., even in attacks against different victims or on different days. To further investigate this observation, we analyzed amplifiers seen at our victim system in self-attacks from a few example booter services over time, shown in Figure 5.1. The entries on the heat maps show the ratio of abused amplifiers that were shared per booter and attack protocol on two consecutive days each. With the exception of DNS, there is a high level of overlap for attacks based on NTP, CharGen, and SSDP, suggesting that booters reuse their set of amplifiers for a protocol for some time. The low overlap for attacks based on DNS is likely caused by frequent rescans to account for the relatively high IP churn rate of DNS amplifiers [80].

In addition, we verified that two simultaneous attacks towards the same victim on different protocols showed little overlap in the sets of honeypots abused. This could indicate that the set of amplifiers might be specific to the protocol, which intuitively can be explained by the small overlap of systems that suffer from amplification vulnerabilities for multiple protocols.

Victim Ports Entropy: While one UDP port determines the amplification protocol (e.g., DNS, NTP, etc.), the other determines the *victim port* on which the victim will receive the reflected responses. Since an attacker has virtually no restrictions on setting the victim port, we expected to observe the two obvious choices: Choosing one victim port per attack, or

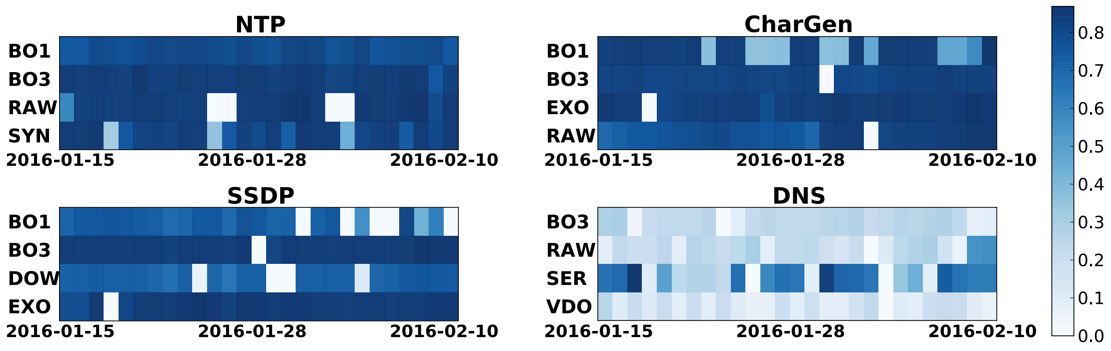


Figure 5.1: Overlap of amplifiers observed by the victim system between consecutive dates.

choosing an individual victim port for every request. Surprisingly, in addition to that, we also observed attacks where requests shared a small number of victim ports. One explanation could be that attackers use multiple threads for attacking, and that they choose a different victim port *per thread*. In addition, we verified that a significant number of booter services actually ask their clients to choose the victim port, giving a reason why the number of source ports is frequently restricted to one.

Time-to-Live Values: The Time-to-Live (TTL) value in the IP packet indicates how many hops a packet has traversed from the attack source to the honeypot. As already observed in early analyses of AmpPot data [S1], for one particular attack, a honeypot will usually only see one (or very few) TTL value(s). We can thus conclude that most attacks likely stem from a single source, which motivates further investigations in finding this particular source sending spoofed traffic. Additionally, the vast majority of requests have a TTL > 230. This suggests that attackers use a fixed initial TTL of 255 in their generated packets, as otherwise we would see a wider distribution.

5.6.2 Distance Function

In order to leverage these observations in a classifier, we next introduce a distance function based on the above features. Given two attack instances A and B, such a function is used to determine how dissimilar the two instances are. For an attack A, we will denote the set of honeypots used by $\mathbf{HP}_A$, the set of victim ports observed by $\mathbf{VPort}_A$, and the set of TTLs received at honeypot hp by $\mathbf{TTL}_{hp,A}$.

To compare honeypot sets, we leverage the well-known Jaccard distance:

$$d_{hp}(A,B) = 1 - \frac{|\mathbf{HP}_A \cap \mathbf{HP}_B|}{|\mathbf{HP}_A \cup \mathbf{IP}_B|}$$

To compare the cardinality of victim port sets, we take the normalized difference:

$$d_{vp}(A,B) = \frac{\big||\mathbf{VPort}_A| - |\mathbf{VPort}_B|\big|}{\max\left(|\mathbf{VPort}_A|, |\mathbf{VPort}_B|\right)}$$

Finally, to compare TTLs, we compute the overlap of their histograms[4]

$$d_{\text{hist}}(S,T) = 1 - \frac{\sum_x \min(S(x),T(x))}{\sum_x \max(S(x),T(x))}$$

and then average this overlap over all honeypots involved in *both* attacks:

$$d_{ttl}(A,B) = \frac{\sum_{hp \in \mathbf{HP}_A \cap \mathbf{HP}_B} d_{\text{hist}}\left(\mathbf{TTL}_{hp,A}, \mathbf{TTL}_{hp,B}\right)}{|\mathbf{HP}_A \cap \mathbf{HP}_B|}$$

From these three sub-functions we compute a weighted average as the overall distance function. We set the weights to $w_{hp} = 5$, $w_{vp} = 1$, and $w_{ttl} = |\mathbf{HP}_A \cap \mathbf{HP}_B|/2$. Note that our methodology is independent from the weights and an analyst may choose any weights according to their needs. We assigned a smaller weight to the victim port feature, as it relies on inputs with little entropy given just three cases: a single victim port, a few victim ports, or many victim ports. For the TTL feature, we assign a higher weight if the two attacks have more honeypots in common, as we assume that coinciding TTLs for *multiple* honeypots have a much higher significance than those for only a single honeypot.

5.7 Honeypot Attack Attribution

We now leverage the aforementioned features to identify which booter has caused which attacks observed at a honeypot. The core idea is to use supervised machine learning techniques to attribute an attack observed at a honeypot to a particular booter service. We will first use our ground truth data set to show the performance and resilience of our classifier in various situations. Afterwards, we will apply the classifier to the entire data set of attacks collected by the honeypots.

5.7.1 Description

Finding the true origin of an amplification attack is a non-trivial problem, because—from the reflector's perspective—all packets carry spoofed headers. However, based on our observation that attacks from the same booter service exhibit similar features, we can cast attack attribution as a classification problem. The collected self-attack data set can be used for training and validating a classifier. We decided to use a *k-Nearest Neighbor* (k-NN) classifier for two reasons. First, k-NN provides some resilience against imbalanced classes. This is essential as the number of attacks our honeypots could observe varies strongly between booters. More importantly though, k-NN is *inspectable* as its classification decisions can be easily retraced. In k-NN, to determine the label of an instance, the set of its k nearest neighbors is computed. Next, every neighbor casts a vote for its own label, and finally the instance is given the label of the majority of its neighbors. Overall, this allows us not only to label an unknown attack, but also provide some justification for the attribution.

[4]To account for fluctuation in TTLs due to route changes, we apply smoothing to the histograms using a binomial kernel of width 6, which corresponds to a standard deviation of $\sigma \approx 1.22$.

Additional care has to be taken, as our training data set is not exhaustive and may miss data for some booters. That is, not all attacks can be attributed to a booter that we know. Therefore, we use a cutoff threshold t to introduce a label for an `unknown` classification result. When classifying an item i, we only consider the k nearest neighbors that can be found in the neighborhood of radius t centered around item i. If no item from the training data set lies within this neighborhood, the item i is assigned the label `unknown`. To find a well-suited and conservative threshold, we analyzed our ground truth data set using our distance function and hierarchical clustering. From those clusters, we then computed the average distance between attacks within a cluster and took the 95th percentile over all. This results in $t = 0.338$ for DNS and $t = 0.236$ for NTP.

Furthermore, as shown in Section 5.6.1, booters rescan on a regular basis to find new lists of amplifiers. To reflect this during classification, we only consider elements from the training data set no more than 7 days apart, which approximately corresponds to the maximum rescan frequency we observed for booters.

When using k-NN, the choice of k is highly critical for the performance of the classifier. One common approach is to learn the value of k from the training data set using *n-fold cross-validation* (CV). In n-fold CV, the training data set is partitioned into n equally sized sets. Then, the classifier is trained on $n-1$ of these sets, and the final set is used for validation. This process is repeated n times, until every set has been used as the validation set once. For finding k we thus perform 10-fold CV for all $k \in \{1, 3, 5\}$ as part of the training phase of the classifier. We restrict k to odd values to avoid ties in the voting phase. We only consider $k \leq 5$, because about $2/3$ of the clusters contain less than five attacks.

To assess the performance of our classifier, we first define the *false positive rate* (FPR), *precision* and *recall* metrics, as well as *macro-averaging*. Intuitively, the FPR for a label l_i (in our case, a particular booter) is the fraction of elements that were incorrectly assigned the label l_i while their true label was *not* l_i. In a similar vein, precision is the ratio with which the classifier was correct when assigning label l_i, while recall is the ratio with which the classifier is able to re-identify elements with true label l_i. Let tp_i be the number of items *correctly* classified to have label l_i (*true positives*), let tn_i be the number of items *correctly* classified to *not* have label l_i (*true negatives*), let fp_i be the number of items *incorrectly* classified to have label l_i (*false positives*), and let fn_i be the number of items *incorrectly* classified to *not* have label l_i (*false negatives*). Then the FPR is defined as $\mathsf{fpr}_i = \mathsf{fp}_i/(\mathsf{fp}_i + \mathsf{tn}_i)$, precision as $\mathsf{p}_i = \mathsf{tp}_i/(\mathsf{tp}_i + \mathsf{fp}_i)$, and recall as $\mathsf{r}_i = \mathsf{tp}_i/(\mathsf{tp}_i + \mathsf{fn}_i)$. To compute overall performance measures from these per-class metrics, we employ macro-averaging, i.e., first computing fpr, p, and r *per class* and averaging the respective results afterwards, as this will avoid bias due to imbalance in our ground truth data. Thus booters for which we were able to collect more data points do *not* influence the results more strongly.

When optimizing the parameter k, we use the macro-averaged FPR as the performance measure. However, since we strongly prefer mislabeling an attack as `unknown` over incorrectly attributing it to a wrong booter, we only weigh the `unknown` label with $1/8$.

5.7.2 Validation

To validate our classifier, we defined three experiments on our labeled self-attack data set: First, we conducted 10-fold CV to assess how well our classifier can correctly attribute attacks (E1). Second, to estimate how well our classifier deals with attacks from booters *not* contained in the training data set, we used leave-one-out CV on the booter level (E2). This means that the attacks from all but one booter constitute the training set, and all attacks from the omitted booter are used for validation, checking if these attacks are correctly labeled as `unknown`. Third, we were also interested in the performance of classifying attacks in real-time (E3), i.e., training only on labeled observations *prior* to the attack.

Table 5.3 and Table 5.4 show, respectively, the results for DNS and NTP. For each experiment we give the percentage of attacks correctly attributed to its booter, the percentage of attacker the classifier labeled as `unknown`, as well as the percentage of attacks that were misclassified, along with their putative label. Additionally, the first column states the number of attacks contained in our data set[5]. Note that in the second experiment (E2) the classifier is trained on the entire data set *except* the corresponding booter; hence the classifier is correct when assigning the `unknown` label in this case.

In the 10-fold CV (E1) our DNS classifier correctly attributed 78% or more of the attacks for each booter. Exceptions are the cases of `EXI` and `VDO`, for which our data set only contains a single attack, which naturally cannot be attributed correctly due to lack of training data. All the remaining attacks were labeled as `unknown`. In fact, the DNS classifier never attributed an attack to a wrong booter in all three experiments.

This is especially remarkable in the leave-one-out scenario (E2), when the classifier was not trained on data for one of the booters. Even in this case our classifier did not lead to false accusations, showing the resilience of the classifier against attacks stemming from booters *not* contained in the training set. Of course, this resilience comes at the cost of higher false negative rates in the other experiments (E1 & E3), as we prefer the classifier to label an attack as `unknown` over attributing it to the wrong booter. This could possibly be alleviated by obtaining more training data per booter.

The last experiment (E3) simulates the performance of the classifier in a real-time scenario, i.e., when classifying an attack only based on training data that was obtained *prior* to the attack. In contrast to this, the first experiment (E1) measured the performance when classifying attacks after the fact. Since booters regularly rescan for amplifiers and update their set of amplifiers accordingly, our classifier will achieve a performance worse than in the first experiment (E1). However, even in the real-time attribution setting, we could still attribute at least 67% of all attacks without any incorrect attributions. The loss compared to E1 can be explained by the fact that the first attack of a booter can never be correctly classified due to lack of prior training data.

In the case of NTP, we achieved an overall attribution rate of 78% or more in the 10-fold CV (E1) for most booters, with the exception of those which occur only once in the data set. Remarkably, the cases of `EXI` and `SYN` show that the classifier also performs reasonably well even for small amounts of training data. The NTP classifier generates misclassifications.

[5]This effectively provides the entire confusion matrix for each experiment.

		10-fold CV (E1)			leave-one-out (E2)		real-time (E3)		
	samples	correct	unknown	wrong	unknown	wrong	correct	unknown	wrong
	(#)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)
BAN	10	90	10		100		70	30	
EXI	1	0	100		100		0	100	
RAW	49	82	18		100		67	33	
SER	11	82	18		100		73	27	
ST1	10	100	0		100		70	30	
ST2	18	78	22		100		67	33	
VDO	1	0	100		100		0	100	
avg	**14.3**	**62**	**38**		**100**		**50**	**50**	
avg$_{\#>1}$	**19.6**	**86**	**14**		**100**		**69**	**31**	

Table 5.3: Classifier Validation Results (DNS)

		10-fold CV (E1)			leave-one-out (E2)		real-time (E3)		
	samples	correct	unknown	wrong	unknown	wrong	correct	unknown	wrong
	(#)	(%)	(%)	(%)	(%)	(%)	(%)	(%)	(%)
AUR	28	100	0		100		89	11	
BAN	15	87	13		100		67	33	
BO1	40	78	23		100		57	43	
BO2	1	0	100		100		0	100	
BO3	12	100	0		100		92	8	
CRI	27	96	4		74	NET: 26	81	19	
DOW	21	90	10		100		76	24	
EXI	4	50	50		100		25	75	
NET	1	0	0	CRI: 100	0	CRI: 100	0	0	CRI: 100
RAW	78	99	1		100		88	12	
SER	16	100	0		100		88	13	
ST1	19	100	0		100		84	16	
ST3	18	94	6		100		78	22	
ST4	24	100	0		100		92	8	
SYN	5	80	20		100		60	40	
THU	1	0	100		100		0	100	
VDO	63	100	0		100		97	3	
avg	**21.9**	**75**	**19**	**6**	**93**	7	**63**	**31**	**6**
avg$_{\#>1}$	**26.4**	**91**	**9**		**98**	2	**77**	**23**	

Table 5.4: Classifier Validation Results (NTP)

However, this only stems from a few attacks by NET and CRI, which exhibit precisely the same characteristics. While we suspect that NET and CRI share the same infrastructure, we were not able to verify this assumption by leveraging layer 7 attacks (as done previously for RAW and WEB). The same two attacks are also the cause for the only misclassifications in the leave-one-out scenario (E2), as about a quarter of attacks from CRI were attributed to NET, when the classifier was not trained on data from CRI. In the real-time scenario (E3), the NTP classifier attributed over 76% of the attacks in most cases, even outperforming the DNS classifier. Since NTP experiences less amplifier churn, booters can use the same amplifier set for a longer period of time, i.e., an attack is more likely to use a set of amplifiers for which the classifier already observed a training sample. A notable exception here is B01, for which only 57% of the attacks could be attributed, despite the large number of attacks contained in the data set. This indicates that B01 performs rescans more frequently than other booters.

Averaging over booters for which the data set contains more than one attack, our classifier achieves a macro-averaged precision of 100.00% and a recall of 86.25% in E1 for DNS, and 99.74% and 91.01% for NTP, respectively. In the case of real-time attribution (E3), the precision stays similarly high (100.00% for DNS, 99.69% for NTP), while the recall drops to 69.35% (DNS) and 76.73% (NTP).

5.7.3 Attribution

After validating the classification mechanism, we now turn to applying it to our entire data set of attacks observed at the honeypots (excluding the self-attacks). Due to their low entropy, we excluded attacks that were only observed by a single honeypot. This left 266 324 NTP-based and 161 925 DNS-based attacks. For both we trained our classifier on all self-attacks collected from December 9 to February 10.

Our NTP classifier attributed 38 520 attacks (14.46%) to one of the booters it was previously trained on and our DNS classifier attributed almost a third of all attacks (49 297, 30.44%) to a booter. Note that not all attacks observed at the honeypots have to be caused by booters; they can also be caused by malevolent parties that do not offer their attack capabilities on an online platform. Furthermore, since we only trained our classifier on a limited set of booters, our classifier cannot possibly achieve a classification rate of 100%. Still, attributing a considerable amount of attacks to the booters of our training set indicates that the booters we considered are used very actively.

5.8 Discussion

We now discuss potential ways to evade our attribution implementation and describe general limitations of our approach that we have not discussed so far.

5.8.1 Evasion

While our attribution methods have proven to work well as of now, they may be susceptible to evasion attempts by miscreants. A *mimicry* attacker could try to be attributed as someone else by following our methodology, i.e., learning the attack profile of another booter and copying its behavior. For example, she could use the same set of reflectors as the other booter for her own attacks. However, this involves a significant increase in terms of effort in comparison to Internet-wide scans. In addition, our TTL-based features are much harder to copy, as they encode the location of the booter service and are subject to changes for other booter locations. While such mimicry attacks are possible [10], given the complexity and overhead, we do not believe that attackers trying to trigger a *false* attribution constitute an actual risk in practice. For similar arguments, attackers that share lists of reflectors with each other would partially poison our analysis, but again TTL-based attribution may be safe against this.

An *evasive* attacker could try to evade our classification mechanisms. Attackers have full control over the traffic they generate, and thus could add noise. For example, one could randomize the set of reflectors used in the attacks, or spoof the initial TTL value within a range of possible values. It is unclear if a classifier could still keep up with such evasion attempts, but it may be possible to add additional features to enrich the classification, such as other characteristics (e.g., IP packet IDs, DNS transaction IDs), as those have shown characteristic patterns even if they were randomized [S1]. In addition, honeypots that selectively respond to scan requests may survive such randomization [P1]. Even if attackers randomize the set of reflectors, any subset will still be a subset of a unique mapping to a scanner. Lastly, randomizing the traffic does also incur a performance overhead to attackers, as they cannot reuse pre-generated packets.

Finally, as in the previous chapter, attackers may try to map out our honeypot amplifiers. Thus the discussion from Section 4.9.3 also applies here.

5.8.2 Limitations

Our in-the-wild experiments faced some limitations, as discussed in the following:

Honeypot Coverage: Regardless of our attempts to maximize the coverage of the honeypots, they missed significant fractions of the self-attacks, especially for SSDP and CharGen. This can be addressed by framing larger emulated responses to make the honeypots more attractive to attackers. The coverage for two of the main protocols, DNS and NTP, was significant, though, covering about 57% of the self-attacks. We therefore argue that our results are representative at least for these two protocols. In addition, there is no limitation of our methodology that would restrict its applicability to the two well-tested protocols.

Multi-Source Attribution: We assumed that attacks are caused by single sources (booters). If botnets launched amplification attacks, our features (e.g., TTL) would be unstable. To give an upper bound of attacks launched by botnets, we searched for attacks with several TTL values, as this—among other reasons—might be caused by distributed traffic sources. Less than 9.5% of attacks at the honeypots show more than 2 TTL values at a honeypot.

Other Attacks: Other types of DDoS attacks, such as SYN flooding or HTTP-based attacks, do not use reflectors and are thus not traceable with our proposed methods. Note that amplification attacks constitute the most common bandwidth exhaustion attack. This is also demonstrated by the fact that all booters advertise amplification attacks, while support for other attack types (e.g., HTTP-based attacks) is far less popular. To put things into perspective: we observed more than 8 900 amplification attacks per day.

5.9 Related Work

Our work was motivated by various research papers that shed light onto booter services using forensic analyses. Karami and McCoy were the first to monitor such booter services, studying the adversarial DDoS-As-a-Service concept [71] and observing that booters are a source for amplification attacks. Similarly, Santanna et al. analyze leaked databases and payment methods of 15 booters [123]. Related to our idea to fingerprint booters, Santanna et al. performed self-attacks of 14 booter services and also observed that the set of reflectors chosen by booters may have overlap across attacks [124]. We build upon this observation, find further correlations for attacks of booter services, and propose to use theses for attack attribution. Karami et al. [72] provide a detailed view on the subscribers and victims of three booters. They provide early attempts to map the infrastructures of booters, but do not perform any kind of attribution between attacks and booters or infrastructures.

Wang et al. [159] have studied the dynamics of attack sources of DDoS botnets, showing distinct patterns per botnet. While the authors provide first results that might enable them to predict future attack sources, they do not further investigate this matter. Our work is different in motivation and techniques in multiple respects. First, booters follow a completely different methodology than DDoS botnets, which rarely use IP spoofing. Second, we can leverage the observation that attackers scan for “attack sources” (amplifiers). Third, we perform attack attribution rather than prediction.

In the previous chapter, Chapter 4, we showed how to uncover the *scan infrastructures* behind amplification DDoS attacks, which in some cases could also be identified to be the attacking infrastructure. Compared to the approach presented in this chapter, there are key differences both in the goal and the methodology: While there we used probabilistic reasoning to identify the scanners that provide the necessary reconnaissance for attacks, here we use machine learning techniques to link attacks to the originating booters. Moreover, both approaches serve different demands: Identifying scanners aids in adding pressure on providers to cease illegal activities, whereas linking attacks to booter services helps to generate forensic evidence, which can prove useful in prosecution.

5.10 Conclusion

Our work presented the first deep exploration of techniques for attributing amplification DDoS attacks to booter services. We present two precise attribution techniques based on carefully chosen features as part of a k-NN classifier. In order to evaluate the effectiveness of our techniques, we subscribed to a small set of booter services and launched self-attacks to collect a ground truth set of attack-to-booter-service mappings. We discuss the ethical framework used to collect this data set, which is similar to that of a previous study [72].

Our honeypot-driven technique attributes DNS and NTP attacks with a very high precision of over 99% while still achieving recall of over 69.35% in the most challenging real-time attribution scenario. Further analysis has revealed that 25.53% (49 297) of the observed

DNS attacks can be attributed to just 7 booter services and 13.34% (38 520) of the NTP attacks can be attributed to 15 booter services. We have shared these findings with law enforcement agencies to help them prioritize legal actions against the wealth of booter services.

6
BGPeek-a-Boo: Active BGP-based Traceback for Amplification DDoS Attacks

6.1 Motivation

Towards the goal of a general attack traceback mechanism, in this chapter we introduce BGPEEK-A-BOO, a BGP-based approach to trace back amplification attacks to their origin network. BGPEEK-A-BOO monitors amplification attacks with honeypots and uses *BGP Poisoning* to temporarily shut down ingress traffic from selected Autonomous Systems. By systematically probing the entire AS space, we detect systems forwarding and originating spoofed traffic. We then show how a graph-based model of BGP route propagation can reduce the search space, resulting in a 5× median speed-up and over 20× for 1/4 of all cases. BGPEEK-A-BOO achieves a unique traceback result 60% of the time in a simulation-based evaluation supported by real-world experiments.

6.2 Problem Description

In the previous chapters we introduced techniques for attack traceback that could identify either the scanner (Chapter 4) behind an attack or the booter service (Chapter 5) for the subclass of attacks launched from known services. However, not all attacks are launched from booter services, and a direct overlap between the scanning and the attacking infrastructure could only in some cases be confirmed. On the other hand, as noted in Chapter 3, general approaches to trace back IP spoofing [70] are mostly based around the idea of packet marking [140, 45, 43, 132, 14, 54], where routers encode path information in the packet header, or collecting flow telemetry data [137, 138, 87, 75, 147]. However, both require the cooperation of a large number of routers along the path and thus a widespread deployment on the Internet—something we have not seen despite these approaches being known for over a decade.

6.3 Contributions

In this chapter we propose BGPEEK-A-BOO, a novel approach to trace back amplification attacks that requires neither cooperation of on-path routers nor knowledge of potential attack sources. The main insight behind our approach is as follows: While attackers may spoof IP level information, they are usually tightly coupled to a given spoofing-capable network location. Packets sent by the attacker are thus bound to the routes chosen by their network provider. This allows us to use the Border Gateway Protocol (BGP) to identify the Autonomous System (AS) that emits the spoofed attack traffic, which constitutes a fundamental step towards fighting these attacks: Once identified, prosecutors can contact the AS operators to investigate the perpetrators behind the attack, which must be customers of the AS. Further, the spoofing AS can be pressured into implementing egress filtering by its peers, similar to what happened to McColo in 2008 [68, 29].

BGPEEK-A-BOO, shown in Figure 6.1, consists of a number of AMPPOT honeypots, which are organized in multiple /24 prefixes, and a BGP router that can advertise routes for these prefixes. As discussed in Section 2.2, AMPPOT emulates services that are vulnerable to

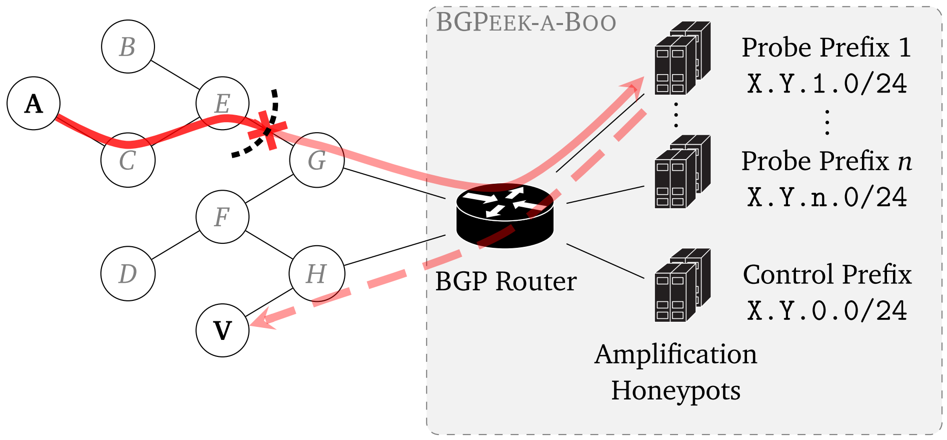


Figure 6.1: BGPEEK-A-BOO overview. After poisoning E, the attack towards V is no longer observed at the amplification honeypots and must therefore originate from either A, B, C, or E.

amplification in order to be selected as reflectors in amplification attacks [S1]. During attacks, the honeypots will then receive spoofed traffic sent by the attacker. Through *BGP Poisoning* we can then exclude certain ASes from propagating routes towards our system. In particular, depriving the attacker of a route causes the spoofed traffic to either switch to an alternative route, which may be observed by a change in TTL values at the honeypots, or to cease entirely. Building on this observation, we systematically probe ASes to uncover those involved in forwarding attack traffic—eventually leading us to the spoofing AS itself.

In a second step, we show how AS relationship data can be used to drastically limit the search space. For this we build a *BGP flow graph* that captures how BGP advertisements propagate and analyze which systems are *reachable* and *dominated* by others. We find that both algorithms achieve a perfect attribution result 100% of the time in an idealized, and still over 60% in a more realistic simulation. Our naive algorithm requires a median of 549 BGP Poisoning steps (91.5 hours) for traceback, while our graph-based algorithm improves this to 98.5 steps (16.4 hours), with 25% of cases even terminating in at most 29 steps (4.8 hours). An 8-fold parallelization of our methodology reduces the median traceback duration to less than an hour.

In summary, our contributions are the following:

1. We propose a novel approach for AS-level traceback of IP spoofing by leveraging BGP Poisoning. Our approach requires neither cooperation of external parties nor a priori knowledge about the attacker.

2. We present two traceback algorithms, showing that BGP-based traceback is feasible in principle and can be greatly sped up when augmented with AS relationship data.

3. We provide an extensive simulation-based evaluation, measuring the influence of various parameters on performance and correctness. We confirm our simulator through real-world experiments using the PEERING BGP testbed [128] and RIPE Atlas [117].

6.4 Background

In this chapter we propose to use BGP Poisoning to identify the origin of spoofed traffic, motivated by the prevalent use of IP spoofing in UDP amplification attacks.

6.4.1 The Border Gateway Protocol (BGP)

The Internet is often described as a "network of networks", as it comprises thousands of so-called *autonomous systems* (ASes). Every AS is a network under the control of a single entity. An AS is usually responsible for a number of IP prefixes and can be identified by its unique *AS number* (ASN).

BGP [115] enables routing between ASes. In BGP, ASes exchange routing information with their neighbors through route advertisements (also called announcements). Each route advertisement describes a path of ASes (AS_PATH) via which a certain IP prefix may be reached. When a BGP router receives an advertisement for a prefix, it first checks if it already knows a *better*[1] route for that prefix. If it does, the new route is only kept as a fallback. Otherwise, the router prepends its own ASN to the AS_PATH and advertises this new route to its neighbors. Between two BGP routers, a new advertisement for a prefix also implicitly withdraws the old route advertised for that prefix. When routing traffic, packets are forwarded according to the best known route for the most specific prefix covering the traffic's destination. We assume that the attacker does not control an entire AS, but is a customer of an RFC-compliant AS.

6.4.2 BGP Poisoning

BGP detects and prevents loops [115]. Before considering new advertisements, routers check if their own ASN is already included in the AS_PATH. If so, the new advertisement will be considered as a withdrawal only and no longer propagated to the AS's neighbors. A side-effect of loop detection is that it enables *BGP Poisoning*. By crafting the AS_PATH to include other systems' ASNs, loop detection can intentionally be triggered at these other systems. Specifically, to trigger loop detection at ASes $X_1, \ldots, X_n$, an AS A may send an advertisement with

$$\texttt{AS\_PATH} = (A, X_1, \ldots, X_n, A)$$

If any AS $\in X_1, \ldots, X_n$ receives this advertisement, it will find itself already present in the AS_PATH, consider this a "loop", and handle it as a withdrawal subsequently. The first A ensures that A's neighbors correctly see A as the next on-path AS, while the last A ensures that the prefix is still seen as originating from A (as required, e.g., for Route Origin Validation [84, 85]). We will call A the *poisoning AS* and X_1 through X_n the *poisoned ASes*. Despite its negative name, BGP Poisoning does not imply malevolence—after all, dropping advertisements can only impede reachability of the *poisoning* AS, but not others. On the contrary, since it gives operators a way to control *in*bound traffic paths, its utility has been proven for many traffic engineering tasks [73, 74, 135].

[1]according to its operator defined policies

6.5 BGP-based Traceback

As noted in Section 6.4.2, BGP Poisoning can be used to discard route advertisements at other ASes. In this section we show how the resulting side-effects can be leveraged to find the origin AS of spoofed attack traffic and present our traceback system BGPEEK-A-BOO.

6.5.1 Poisoning for Traceback

Assume an AS A is sending spoofed traffic to a reflector and receives a poisoning advertisement for the reflector's prefix. Since the AS will handle this advertisement like a withdrawal, it will remove its routing information for that prefix. However, without routing information it can no longer send traffic to the reflector. The reflector will hence stop receiving traffic from A.

A similar observation can be made for ASes *forwarding* traffic to the reflector. Assume an AS F is normally forwarding spoofed traffic from A to the reflector and receives a poisoning advertisement. Next to losing the ability to send traffic to the reflector, F also has to withdraw any routes for that prefix it had advertised to its neighbors. Thus A can no longer route traffic to the reflector via F. If no alternative path from A to the reflector exists that circumvents F, then A again loses connectivity and traffic at the reflector will stop. However, even if an alternative path exists, this case might be observable at the reflector: Unless the IP hop count along both paths is exactly the same, the traffic's TTL value at the reflector will change.

This allows us to check whether some AS X was *on-path* of a traffic flow: If poisoning X causes the traffic to stop or its TTL value to change, X *was* on-path. Furthermore, if traffic stopped, the origin AS has no alternative routes avoiding the poisoned AS X.

6.5.1.1 Default Routes

In practice, some ASes can still route traffic even for prefixes that they have no explicit routing information for. These *default routes* can be either configured statically or an upstream AS can advertise itself as the next hop for a large prefix (e.g., `0.0.0.0/0`). In these cases, the poisoned AS will not loose connectivity entirely, but switch to its default route. As before, such a route change can result in an observable TTL change. Furthermore, since a default route cannot be advertised with a more specific prefix than the original route, BGP will avoid paths including a default route whenever possible.

6.5.1.2 Combined Probes

The on-path check can also be performed for multiple ASes simultaneously using a combined poisoning advertisement. If the poisoning of an AS leads to a stop in traffic, poisoning additional ASes cannot undo this effect. If, however, it leads to a TTL change only, poisoning additional ASes may cause further re-routing or even eliminate alternative paths. While this could negate the TTL change under specific circumstances[2], it results in additional traffic stops in many cases.

[2]if it causes traffic to take a path with the exact same IP hop count as the original path

Ultimately though, a combined probe can only tell whether *any* of the poisoned ASes were on-path. To find the exact on-path AS within the probed set, we can use a binary search approach, iteratively splitting the probing set in half and repeating the probing for each half.

6.5.1.3 Active Measurements

This technique can be further supplemented by active measurements. By provoking replies from a host in the measured AS (e.g. through ICMP Pings or TCP SYNs[3]), we can observe the effect of poisoning on the AS. If the replies to our active measurements stop under poisoning, but the spoofed traffic continues (or vice versa), we can conclude that the measured AS was *not* the spoofing source.

6.5.1.4 Probes in BGPeek-a-Boo

BGPeek-a-Boo uses all but one of its prefixes as *probe prefixes* to probe network responses to poisoning advertisements. The remaining prefix is designated as the *control* prefix and will only be advertised regularly (non-poisoned). We will refer to honeypots in probe prefixes and the control prefix as *probe honeypots* and *control honeypots* respectively.

To probe an AS, the BGP router of BGPeek-a-Boo sends a poisoned route advertisement for a probe prefix that receives attack traffic. Once the routes have stabilized, it records the impact on the attack traffic and on pings. If the attack is also observed by some control honeypots, impact can also be measured by comparing traffic between the probe and control honeypots.

6.6 A naive Traceback Approach

Using these measures we propose the naive traceback algorithm depicted in Algorithm 6.1. The algorithm simply loops over all ASes (set $\mathscr{A}$) in chunks of size at most n and considers each chunk as a combined probe $\mathscr{P}$.

The actual probing is performed by Probe, which sends out a poisoned advertisement for the ASes in $\mathscr{P}$, performs the active measurements, and returns the observed effects. It returns the effect on the attack traffic (r_{passive}) and the active measurement results ($\vec{r}_{\text{active}}$). r_{passive} can be either `NO_EFFECT` if no change was observed, `TTL_CHANGE` if we observed a TTL change, or `STOP` if traffic has stopped entirely. Similarly, $\vec{r}_{\text{active}}$ is a vector with one component (either `NO_EFFECT` or `STOP`) per probed AS.

If poisoning of $\mathscr{P}$ shows an effect on the attack traffic, the probe is then narrowed down to find the exact AS(es) that caused this effect. For this, first, all probed ASes that show an inconsistent behavior in their *active* measurements are discarded from the probe (Update). If this already reduces the probe to a single consistent AS, it is then added to the candidate set $\mathscr{C}$ as a confirmed on-path AS. Otherwise, the probe is split in half (Split) and the probing is repeated for each half recursively. Once all ASes have been probed that way, the final set of confirmed on-path ASes $\mathscr{C}$ is returned.

[3]suitable candidates and ports could be found through Internet-wide scanning or by leveraging a search engine such as Shodan[152]

Algorithm 6.1 Naive traceback algorithm

```
procedure NaiveTraceback($\mathscr{A}$, $A$, $n$)
    $\mathscr{C} \leftarrow \emptyset$                                         ▷ candidates
    for block $\mathscr{P}$ in $\mathscr{A}$ of size $\leq n$ do
        ProbeAndUpdate($\mathscr{P}$)
    return $\mathscr{C}$

procedure ProbeAndUpdate($\mathscr{P}$)
    $r_{\text{passive}}, \vec{r}_{\text{active}} \leftarrow$ Probe($\mathscr{P}$)
    if $r_{\text{passive}} \neq$ NO_EFFECT then
        $\mathscr{P} \leftarrow$ Update($\mathscr{P}, r_{\text{passive}}, \vec{r}_{\text{active}}$)
        if $|\mathscr{P}| = 1$ then
            $\mathscr{C} \leftarrow \mathscr{C} \cup \mathscr{P}$                  ▷ AS in $\mathscr{P}$ was on-path
        else if $|\mathscr{P}| \geq 2$ then
            $\mathscr{P}_1, \mathscr{P}_2 \leftarrow$ Split($\mathscr{P}$)            ▷ “Binary Search”
            ProbeAndUpdate($\mathscr{P}_1$)
            ProbeAndUpdate($\mathscr{P}_2$)

procedure Update($\mathscr{P}, r_{\text{passive}}, \vec{r}_{\text{active}}$)
    if $r_{\text{passive}} =$ STOP then
        $\mathscr{P}_{\text{inconsistent}} \leftarrow \{X \in \mathscr{P} \mid \vec{r}_{\text{active}}[X] \neq \text{STOP}\}$
    else
        $\mathscr{P}_{\text{inconsistent}} \leftarrow \{X \in \mathscr{P} \mid \vec{r}_{\text{active}}[X] = \text{STOP}\}$
    return $\mathscr{P} \setminus \mathscr{P}_{\text{inconsistent}}$
```

6.6.1 Runtime Analysis

The runtime of this traceback approach is greatly dominated by the probing step, which involves sending out a poisoned advertisement and performing active measurements. After a new advertisement, it may take several minutes for routes to settle [82, 74], only after which active measurements can be performed. In addition, several BGP mechanisms further limit the rate at which routers may send out new advertisements (Section 6.9.1). Therefore, realistically, advertisements cannot be made much faster than once every ten minutes.

We will thus count the number of required probing steps to analyze the traceback runtime. At chunk size n and a total of N ASNs in $\mathscr{A}$, the naive traceback algorithm from Algorithm 6.1 takes $\lceil \frac{N}{n} \rceil$ steps to test each chunk once, plus an additional $2 \log_2 n$ steps for every on-path AS to reduce the chunk it is contained in down to a single AS. For example, the AS65000 BGP Routing Table Analysis Report [7] lists roughly 66000 active ASes for the end of 2019 and an average AS path length of 5.5. With a chunk size of $n = 128$ this thus gives an average of 593 steps total, or about 4 days and 3 hours at 6 advertisements per hour.

Given that spoofing sources, such as Booter services, are active for extended periods of time (see Section 6.8.3.4) and reuse the same amplifiers for a week or longer [P2], this shows that BGP-based traceback is feasible in principle.

6.6.2 Discussion

While this algorithm is intuitive and requires no external knowledge of AS properties or relationships, it comes with two main drawbacks:

1. It only returns an unordered set of on-path ASes, which still leaves the exact path and origin unknown.
2. It “wastes” a lot of time poisoning off-path ASes that could potentially be avoided, as it effectively conducts an exhaustive search over the entire AS space.

6.6.2.1 On-Path Ordering

Ideally, one would like to find the true origin AS of the spoofed traffic, or at least the on-path AS closest to it that can still be discovered. However, through probing we can only tell *whether* an AS was on-path or not, but not its position along the path.

While at first glance it seems that this problem could be solved by comparing TTL values received from hosts located in these ASes, this is not necessarily true: Although the AS level path should be the same for both traffic originating from and traffic forwarded by an AS, the IP level paths (and hence their hop counts) can differ vastly. In a similar vein, one might attempt to infer the on-path order from traceroutes towards hosts in the candidate ASes, mapping the obtained IP level traceroute paths into AS level paths. Barring the problems of mapping IP to AS level paths [170, 66], traceroutes from the vantage point can only reveal paths *towards* other hosts, but not their reverse paths, which we are interested in.

6.6.2.2 Runtime Improvement

Although the estimated runtime does not seem prohibitive, the question still remains whether additional knowledge about ASes, such as their relationships, can be used to achieve effective traceback more efficiently. For example, as a simple optimization the search can be aborted as soon as a stub AS is confirmed to be on-path. Since stub ASes do not provide transit for other ASes, an on-path stub AS must be the one originating the observed traffic. In these cases, such an early termination will reduce the expected runtime to 1/2 of the original algorithm. In the next section, we show how the runtime can be further improved by leveraging AS relationship data as well as the information about alternative path availability one can obtain from probing.

6.7 Flow Graph-based Traceback

Our graph-based traceback algorithm exploits knowledge about the relationship between ASes to limit the search space. For example, if the attack traffic stops under poisoning, we know that the source has no alternative route that avoids the poisoned ASes. To efficiently reason about alternative paths and whether an advertisement might be propagated from one AS to another, we define a so-called *AS Flow Graph*.

6.7.1 AS Flow Graphs

The relationship between two AS is usually classified as either a *customer-provider* (*CP*) or a *peer-to-peer* (*P2P*) relation [53]. In a *customer-provider* relation, one AS (the customer) pays the other (the provider) for transit such that the customer may reach and be reached from the Internet via the provider. In a *peer-to-peer* relation, two ASes agree to transit traffic for their customers to one another, thereby reducing the amount of traffic they would have to pay their provider for otherwise.

The resulting BGP paths are generally assumed to be valley-free [53]: zero or more *customer-provider* edges ("up"), followed by at most one *peer-to-peer* edge ("sideways"), followed by zero or more *provider-customer* edges ("down"). In other words: a *peer-to-peer* or *provider-customer* edge can never be followed by a *customer-provider* or *peer-to-peer* edge, as this would result in a "valley". Note that this property is symmetric and holds for both, the AS-level paths taken by routed traffic as well as the propagation paths of BGP advertisements.

Following this, an AS may receive an advertisement in two states: If the advertisement was received from a customer (i.e., via a *customer-provider* edge), the valley-free assumption does not restrict the edge types that may follow. We will call this state *unconstrained*. On the other hand, if an advertisement was received from either a peer or a provider, it may only be forwarded to customers, but not to other peers or providers. We will thus call this state *constrained*.

We can use a graph to model advertisement propagation that uses *two* nodes per AS, one for each state. Formally, we define this graph $G = (V, E)$ as follows: Every AS A is represented by two vertices, u_A and c_A, representing the *unconstrained* and *constrained* state respectively,

$$V = \bigcup_{A \in \text{AS}} \{u_A, c_A\}$$

We will denote the AS represented by a node x using $\text{asn}(x)$,

$$\text{asn}(x) = A \Leftrightarrow x \in \{u_A, c_A\}$$

When AS A is a provider of AS B, advertisements may only be forwarded "downhill" from c_A to c_B or "uphill" from u_B to u_A. If A and B share a *P2P* relation, then advertisements may only be forwarded "sideways" between A and B at the "peak" of the path, i.e., from u_A to c_B or from u_B to c_A. Finally, advertisements received at u_A may of course also be forwarded to c_A. In total,

$$\begin{aligned} E = &\left(\bigcup_{A,B \in \text{CP}} \{(u_A, u_B), (c_B, c_A)\} \right) \\ &\cup \left(\bigcup_{A,B \in \text{P2P}} \{(u_A, c_B), (u_B, c_A)\} \right) \\ &\cup \left(\bigcup_{A \in \text{AS}} \{(u_A, c_A)\} \right) \end{aligned}$$

This graph then captures all valley-free AS paths, and every path in this graph corresponds to a valid valley-free AS path.

Theorem 1. *Let $\Pi = (X_1,\ldots,X_n) \in \mathrm{AS}^n$ be a valley-free path from AS X_1 to AS X_n and $G = (V,E)$ the flow graph constructed according to Section 6.7.1. Then there exists a path $\pi = (x_1 = u_{X_1},\ldots,x_m = c_{X_n}) \in V^m$ in G.*

Proof. Since $\Pi = (X_1,\ldots,X_n)$ is a valley-free path, it can be split into three (possibly empty) parts: an "uphill" prefix, a single "sideways" peer-to-peer link, and a "downhill" suffix. Formally, $\exists k,l, 1 \le k \le l \le k+1 \le n$ s.t.:

1. $\forall 1 \le i < k : (X_i,X_{i+1}) \in \mathrm{CP}$
2. $k < l \implies (X_k,X_l) \in \mathrm{P2P}$
3. $\forall l \le i < n : (X_{i+1},X_i) \in \mathrm{CP}$

For the first part, we can find a path in G as by construction it holds that $\forall 1 \le i < k$: $(u_{X_i},u_{X_{i+1}}) \in E$. Likewise, for the last part, since $(X_{i+1},X_i) \in \mathrm{CP}$ it holds that $\forall l \le i < n$: $(c_{X_i},c_{X_{i+1}}) \in E$. If the path has a peer-to-peer link, i.e., $k < l$, then $(u_{X_k},c_{X_l}) \in E$. Otherwise, it holds that $X_k = X_l$ and thus $(u_{X_k},c_{X_l}) = (u_{X_k},c_{X_k}) \in E$. Therefore, the path $\pi = (u_{X_1},\ldots,u_{X_k},c_{X_l},\ldots,c_{X_n})$ is in G and satisfies the requirements. □

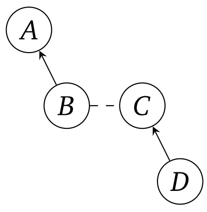
Theorem 2. *Let $G = (V,E)$ be the flow graph constructed according to Section 6.7.1 and $\pi = (x_1,\ldots,x_m) \in V^m$ be a path in G. Then there exists a valley-free path $\Pi = (X_1 = \mathrm{asn}(x_1),\ldots,X_n = \mathrm{asn}(x_m)) \in \mathrm{AS}^n$.*

Proof. W.l.o.g. assume that $x_1 = u_{X_1}$ and $x_m = c_{X_n}$. Since all edges in E are of the form (u_A,u_B), (c_A,c_B), or (u_A,c_B) the path can only have one transition from "unconstrained" to "constrained" nodes, i.e., $\exists 1 \le i < n$ such that

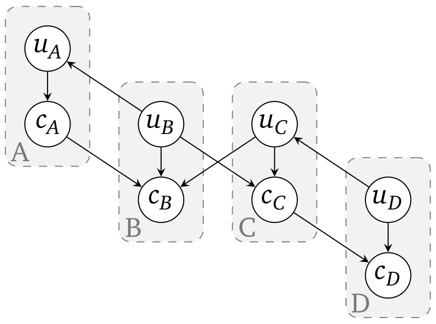
1. $\forall 1 \le j \le i : x_j \in \{u_A | A \in \mathrm{AS}\}$
2. $\forall i < j \le n : x_j \in \{c_A | A \in \mathrm{AS}\}$

Therefore, for $j < i$ it holds that $(\mathrm{asn}(x_j),\mathrm{asn}(x_{j+1})) \in \mathrm{CP}$, as otherwise there would be no edge in G between them. Likewise, for $j > i$ it holds that $(\mathrm{asn}(x_{j+1}),\mathrm{asn}(x_j)) \in \mathrm{CP}$. Finally, for the edge $(x_i,x_{i+1}) = (u_{X_k},c_{X_l})$ it must hold that either $X_k = X_l$ or that $(X_k,X_l) \in \mathrm{P2P}$. Therefore, $\Pi = (\mathrm{asn}(x_1),\ldots,\mathrm{asn}(x_m))$ follows the definition of a valley-free path, with the exception that it may still contain loops. However, if Π contains a loop, i.e., $\exists 1 \le i < j \le n$ s.t. $\Pi = (X_1,\ldots,X_i,\ldots,X_j,\ldots X_n)$ with $X_i = X_j$, one can easily construct a loop-free path $\Pi' = (X_1,\ldots,X_i,\ldots X_n)$ by omitting positions $i+1$ through j. Note that removal of loops does not affect the path's endpoints, and thus the resulting AS path Π' is a valley-free path from $\mathrm{asn}(x_1)$ to $\mathrm{asn}(x_m)$. □

By construction, the orientation of edges denotes the direction of advertisement propagation. For example, an edge from u_B to u_A implies that advertisements received by B may be further propagated to A. However, the inverse graph obtained by reversing all edges is meaningful as well, as it describes all possible traffic flows between ASes: If AS B advertises a route for a prefix to AS A, then A may send traffic destined towards that prefix to B.



(a) Example AS relationships: A is provider of B, C is provider of D, and B and C have a peer-to-peer relation.



(b) The resulting AS flowgraph shows that route advertisements from D may reach C and B, but can never reach A.

Figure 6.2: AS flowgraph example

In the example given in Figure 6.2, A is a provider of B, C is a provider of D, and B and C have a peer-to-peer relation. However, it is not obvious from the relationship graph, whether advertisements from D may reach A. The resulting flowgraph answers this unequivocally: Advertisements from D can reach C ($u_D \rightarrow u_C$) and B ($u_D \rightarrow u_C \rightarrow c_B$), but cannot reach A, since there is no directed path from either u_D or c_D to u_A or c_A.

6.7.1.1 Reachability and Dominance

A poisoned advertisement can affect ASes in two ways: ASes included in the advertisements' `AS_PATH` are affected directly, as they will discard the advertisement due to loop detection. However, this also prevents them from propagating the advertisement further, which, in turn, can affect other ASes. While some of these indirectly affected ASes may still receive the advertisement via alternative routes and thus only experience a route change, others may no longer be able to receive the advertisement at all. To reason about the propagation of (poisoned) advertisements originating from a specific AS A, we can define its AS-specific subgraph:

Definition 1 (AS-specific subgraph). *For the flow graph G and AS A, we define the AS-specific rooted subgraph $G_A = (V_A, E_A, u_A)$ as the subgraph of G rooted at u_A.*

We can then define two relations on this graph, reachability and (joint) dominance, to capture which ASes *might* potentially be affected by a poisoned advertisement and which *will* be inevitably.

Consider an advertisement that is poisoning ASes $P = \{X_1, \ldots, X_n\}$ (with corresponding nodes $p = \{u_{X_1}, c_{X_1}, \ldots, u_{X_n}, c_{X_n}\}$). Another AS Y might only be affected by this advertisement if it can receive advertisements from A via any AS in P[4]. As all paths in the graph G_A describe valid propagation paths for advertisements originating from A, AS Y therefore might be affected if there is a path from any node $x \in p$ to any node $y \in \{u_Y, c_Y\}$.

[4]Under specific circumstances, poisoned advertisements may also induce route changes at ASes that have no connection to a poisoned ASes. We analyze these cases in Section 6.9.4.

Definition 2 (Reachability). *We call a node y* reachable *from another node x, iff there exists a path in G_A from x to y.*

$$\begin{aligned} & y \in \text{reachable}_{G_A}(x) \Leftrightarrow \\ & \quad \exists \pi = (x_1 = x, \ldots, x_n = y) : \\ & \quad\quad \forall 1 \le i < n : (x_i, x_{i+1}) \in E_A \end{aligned}$$

The set $\text{reachable}_{G_A}(x)$ describes all nodes whose traffic towards A *may* be routed via x. The definition of reachable can be trivially extended to sets of nodes by taking their union:

$$\text{reachable}_{G_A}(\{x_1, \ldots, x_n\}) = \bigcup_{i=1}^{n} \text{reachable}_{G_A}(x_i)$$

Therefore, the set of ASes that *might* be affected by an advertisement poisoning nodes p is simply $\text{reachable}_{G_A}(p)$. That is, if poisoning of nodes p causes the TTL values of the attack traffic to change we can infer that the origin was affected and must be an element of $\text{reachable}_{G_A}(p)$.

In a similar vein we can also define nodes that *must* be affected. Intuitively, a node y must be affected if it cannot receive advertisements from A but via nodes in p. Thus, once all nodes in p are removed from the graph, y should be no longer reachable from u_A. In graph theory terms p then constitutes a *u_A-y-vertex-separator* (albeit defined over directed graphs), but can also be seen as a generalization of the concept of *dominators* from control-flow-graph analysis.

Definition 3 ((Joint) Domination). *We call a node y (jointly)* dominated *by a set of nodes $\{x_1, \ldots, x_n\}$ in graph G_A, iff y is reachable from u_A but removing the set $\{x_1, \ldots, x_n\}$ breaks reachability from u_A to y. Formally*

$$\begin{aligned} & y \in \text{dominatees}_{G_A}(\{x_1, \ldots, x_n\}) \Leftrightarrow \\ & \quad y \in \text{reachable}_{G_A}(u_A) \wedge y \notin \text{reachable}_{G'_A}(u_A) \end{aligned}$$

where $G'_A = (V'_A, E'_A, u_A)$ is the subgraph of $G_A = (V_A, E_A, u_A)$ with nodes $V'_A = V_A \setminus \{x_1, \ldots, x_n\}$ and edges $E'_A = \{(x, y) \in E_A \mid x \in V'_A \wedge y \in V'_A\}$.

Hence, the set of ASes that *will* inevitably be affected, i.e., those that will have to switch to default routes or possibly loose connection, by an advertisement poisoning nodes p is $\text{dominatees}_{G_A}(p)$. As noted above, other ASes from $\text{reachable}_{G_A}(p)$ might be affected as well, but for those contained in $\text{dominatees}_{G_A}(p)$ we can be certain.

6.7.2 Flow Graph-based Traceback

We can leverage this flow graph for AS traceback in two ways: First, given a known on-path AS we know that the neighboring on-path AS must also be one of its successors in the graph. Instead of globally searching for the origin AS we can therefore iteratively find all on-path ASes one-by-one by probing the successors of the last on-path AS.

Second, we can use reachability and dominance relations given by the graph to reduce our probing search space: If traffic stops, we know that the origin AS has no alternative paths available and can thus limit our search space to nodes dominated by the probed ASes (which includes the probed ASes themselves). If we observe a TTL change, we can still infer that the origin AS is at least indirectly affected by the probe and can therefore limit our search space to those nodes that are reachable from the probed ASes. We can also use similar inferences on the results of our active measurements: When responses to active measurements for a probed AS stop but the attack traffic does not, we can exclude all nodes dominated by the probed AS from our search space, as those would no longer be able to send traffic as well. Vice versa, if the attack traffic stops but active measurements show that a probed AS is still replying to pings, we can exclude all nodes reachable by the probed AS, as they also could send traffic to us via the probed AS.

6.7.2.1 Graph-based Traceback Algorithm

Combining both of these effects leads to the graph-based traceback algorithm shown in Algorithm 6.2. Given the rooted subgraph $G_A = (V_A, E_A, u_a)$ of the receiving AS A and a probe size limit n, it returns a set of candidate source ASes.

For this, the algorithm maintains a set of source candidates $\mathscr{C}$, which is initially set to all ASes reachable from A and then continuously narrowed down during traceback. Further, it also keeps a logbook $\mathscr{L}$ of ASes that have already been probed and can thus be excluded from further probing, as well as the most recent on-path AS L, which is initially set to A. As long as there are candidates that have not been probed yet ($\mathscr{C} \setminus \mathscr{L}$), a new set of ASes to probe is selected (see Section 6.7.2.2) and passed to PROBEANDUPDATE, which will perform the probing and update the candidate set and logbook accordingly.

As before, probing is performed by PROBE (see Algorithm 6.1). Updating the candidate set is performed in two steps: First, UPDATEPASSIVE reduces the candidate set based on the overall effect on the attack traffic, limiting $\mathscr{C}$ to the set of nodes dominated or reachable by the current probe if traffic stopped or a TTL change was observed. Second, UPDATEACTIVE further narrows down the candidate set and the current probe by finding probed ASes whose active measurements are inconsistent with the overall observation and then removing nodes dominated or reachable by these.

If there has been an effect on the attack traffic and multiple probed ASes exhibit a consistent behavior, they are split in half (SPLIT) and the probing is repeated for each half. If a probe has been narrowed down to a single consistent AS, this AS is set as the most recent on-path AS L and a new probe is selected.

6.7.2.2 Probe Selection

The probe selection routine (PICKPROBE) picks a new probe based on the current candidates $\mathscr{C}$, the ASes that have been probed before $\mathscr{L}$, the last discovered on-path AS L, and the maximum probe size n. As discussed above, the next on-path AS must be a successor of L. We can thus partition our remaining search space of unprobed candidates $\mathscr{C} \setminus \mathscr{L}$ into "layers" according to their distance to the most recent on-path AS L. The next on-path AS should

Algorithm 6.2 Flow Graph-based traceback algorithm

```
procedure TRACEBACK($G_A, n$)
    $\mathscr{C} \leftarrow \{\text{asn}(v) \mid v \in V_A\}$)                ▷ candidates
    $\mathscr{L} \leftarrow \emptyset$                                    ▷ logbook
    $L \leftarrow A$                                      ▷ most recent on-path AS
    while $\mathscr{C} \setminus \mathscr{L} \neq \emptyset$ do
        $\mathscr{P} \leftarrow$ PICKPROBE($\mathscr{C}, \mathscr{L}, L, n$)
        $\mathscr{L} \leftarrow \mathscr{L} \cup \mathscr{P}$
        PROBEANDUPDATE($\mathscr{P}$)
    return $\mathscr{C}$

procedure PROBEANDUPDATE($\mathscr{P}$)
    $r_{\text{passive}}, \vec{r}_{\text{active}} \leftarrow$ PROBE($\mathscr{P}$)
    UPDATEPASSIVE($\mathscr{P}, r_{\text{passive}}$)
    UPDATEACTIVE($\mathscr{P}, r_{\text{passive}}, \vec{r}_{\text{active}}$)
    if $r_{\text{passive}} \neq$ NO_EFFECT then
        if $|\mathscr{P}| = 1$ then
            $L \leftarrow X \in \mathscr{P}$                          ▷ AS $X$ was on-path
        else if $|\mathscr{P}| \geq 2$ then
            $\mathscr{P}_1, \mathscr{P}_2 \leftarrow$ SPLIT($\mathscr{P}$)                  ▷ “Binary Search”
            PROBEANDUPDATE($\mathscr{P}_1$)
            PROBEANDUPDATE($\mathscr{P}_2$)

procedure UPDATEPASSIVE($\mathscr{P}, r_{\text{passive}}$)
    if $r_{\text{passive}} =$ STOP then
        $\mathscr{C} \leftarrow \mathscr{C} \cap \text{dominatees}_{G_A}(\mathscr{P})$
    else if $r_{\text{passive}} =$ TTL_CHANGE then
        $\mathscr{C} \leftarrow \mathscr{C} \cap \text{reachable}_{G_A}(\mathscr{P})$

procedure UPDATEACTIVE($\mathscr{P}, r_{\text{passive}}, \vec{r}_{\text{active}}$)
    if $r_{\text{passive}} =$ STOP then
        $\mathscr{P}_{\text{inconsistent}} \leftarrow \{X \in \mathscr{P} \mid \vec{r}_{\text{active}}[X] \neq \text{STOP}\}$
        $\mathscr{C} \leftarrow \mathscr{C} \setminus \text{reachable}\, G_A(\mathscr{P}_{\text{inconsistent}})$
    else
        $\mathscr{P}_{\text{inconsistent}} \leftarrow \{X \in \mathscr{P} \mid \vec{r}_{\text{active}}[X] = \text{STOP}\}$
        $\mathscr{C} \leftarrow \mathscr{C} \setminus \text{dominatees}_{G_A}(\mathscr{P}_{\text{inconsistent}})$
    $\mathscr{P} \leftarrow \mathscr{P} \setminus \mathscr{P}_{\text{inconsistent}}$
```

then be part of the nearest layer. This ordering also ensures that, should we be unable to find the direct on-path successor of L (e.g., because it exhibits no observable poisoning reaction) we only gradually expand our search scope to later on-path ASes.

Intuitively, to reduce the candidate set as fast as possible, we would like to first probe ASes that have a high impact on the candidate set. While stub ASes can have a high impact if they are on-path (effectively reducing the candidate set to a single entry), statistically speaking, in most cases probed ASes will be off-path ASes. The majority of candidate set reductions will thus occur in UPDATEACTIVE. We therefore rank ASes by the number of candidates that are *reachable* through them, i.e., $\left|\text{reachable}_{G_A}(X) \cap \mathscr{C}\right|$, and then select the n highest ranking ASes as the next probe.

6.8 Evaluation

We next turn to evaluate the runtime and success rate of BGPEEK-A-BOO and compare both proposed algorithms. For this, we were fortunate enough to obtain a temporary ASN and a temporary /22 prefix allocations for research purposes from our regional Internet registry and were granted access to the PEERING BGP testbed [128]. However, PEERING's path length limit unfortunately proved prohibitive for any actual traceback runs of BGPEEK-A-BOO on the Internet: Although a recent study showed that advertisements even with long paths of up to 255 hops are propagated to the vast majority of the Internet [134], we were only able to send advertisements with up to 5 hops via PEERING. Since the first and last of these also had to be set to our own temporary ASN, this left us with an effective probe size of 3. Unfortunately, with this even a single run of our baseline approach would have taken over 157 days to complete. We therefore resort to simulation for a comparative evaluation of our proposed traceback approaches and use the PEERING testbed for supporting experiments.

6.8.1 Simulation Methodology

As noted by previous works [130, 129, 135], a complete and fully accurate model of the entire Internet cannot be obtained, as it would require exact knowledge of peering agreements and router configurations, both of which are usually regarded as trade-secrets and thus generally non-public. Simulation can therefore only be performed over approximate topology data, such as the one regularly published by CAIDA [151], and by making assumptions on router configurations.

Although the simulator by Smith et al. [135] is thankfully publicly available, we found it unsuitable for our use-case as it does model neither default routes nor TTL values along paths. Consequently, we designed our own lightweight simulator.

6.8.1.1 Simulator Design

Our simulator is based around the same AS flow graph described in Section 6.7.1. To model AS paths taken from a source to a destination, our simulator assigns every edge in the graph a random weight between 0 and 100 and then computes the shortest path. This is in line with the regular BGP decision process [115, sec. 9.1], which generally prefers shorter AS paths. The random edge weight hereby models the local preference value a network operator may set to prefer one link over another. As the graph has *two* nodes corresponding to each AS X, u_X and c_X, we ensure that the edge (u_X, c_X) is assigned the weight 0, which also ensures that the resulting AS paths are loop free—a path containing both u_X and c_X can never be shorter than the one that uses the zero-weight (u_X, c_X) edge.

The effect of a poisoning advertisement can then be simulated by temporarily removing the poisoned AS's nodes from the graph, such that they can no longer send traffic to the destination nor forward advertisements to their customers or peers.

In contrast to previous works, our simulator also attempts to model the presence of default routes, which were identified as a major culprit for discrepancies between simulations and experimental results in other BGP-based tools [134]. For this, every AS is randomly marked as either *having* a default route or not with a certain probability. When simulating a poisoning advertisement, those ASes with default routes are not removed from the graph, but instead increase the weights of their incoming edges by 10000. This ensures they are no longer selected as shortest paths, unless no alternative is available—as would be the effect of less-specific prefixes. ASes with default routes also have a chance of using a secondary set of incoming edge weights when being poisoned, as otherwise their default route would always be identical to their regular route.

Finally, our simulator allows mapping AS paths to IP hop counts. For this, every step along the AS path is assigned a random hop count value drawn from a negative binomial distribution, which we found to be a good fit after analyzing traceroute results from RIPE Atlas [117] (see Section 6.8.3.3). In the real Internet, the IP-level path length also depends on which ingress and egress routers are taken. We therefore make this random value also dependent on the preceding and succeeding AS hop.

6.8.2 Results

With this simulator, we conducted multiple experiments to compare our different traceback algorithms in terms of efficacy and efficiency as well as the influence of various parameters, such as the placement of the deployment location, the presence of default routes, and the choice of probe size.

As a baseline setup for our evaluation we placed the deployment location as a customer of the PEERING testbed (AS47065), which fosters comparability with our real-world experiments (Section 6.8.3). We set the default route probability to 40% as a conservative approximation, given that Smith et al. [134] report a default route prevalence between 26.8% and 36.7% in their experiments. Lastly, we picked 128 as a conservative probe size, since they also report successful propagation of paths of length up to 255 [134]. The flow graph used by our simulator was based on the public CAIDA AS relationship dataset for December 2019 [151], which also covered the timeframe when our real-world experiments were performed, augmented by the peering relations listed by the PEERING testbed [128].

Every experiment was repeated 1 024 times, each time with a randomly chosen AS as the traffic's origin. In addition, the simulator was also given a fresh random seed for every run, such that our results are not biased due to a single lucky weight assignment or similar. We use *naive* to refer to the naive algorithm, *naive+* for the variant with early termination, and

	$x \leq 1$	$x \leq 2$	$x \leq 3$	$x \leq 4$	$x \leq 5$	$x \leq 6$	$x \leq 7$	$x \leq 8$
naive	**10**% ±2%	**32**% ±3%	**52**% ±3%	**59**% ±3%	**60**% ±3%	**61**% ±3%	**61**% ±3%	**61**% ±3%
naive+	**28**% ±3%	**46**% ±3%	**57**% ±3%	**60**% ±3%	**61**% ±3%	**61**% ±3%	**61**% ±3%	**61**% ±3%
graph	**58**% ±3%	**62**% ±3%	**65**% ±3%	**66**% ±3%	**66**% ±3%	**67**% ±3%	**68**% ±3%	**68**% ±3%

Table 6.1: Success rate with a final candidate set of size at most x

graph for the graph-based algorithm.

6.8.2.1 Traceback Success

To analyze the efficacy of our proposed traceback, we analyzed how often BGPEEK-A-BOO succeeds in finding the origin. The naive algorithm starts with an empty candidate set and selectively adds on-path ASes. Hence we consider it successful if the true origin is contained in its final candidate set. In contrast, the graph-based algorithm assumes all ASes as candidates initially, but excludes ASes during the traceback run. Therefore, to be successful, the final candidate set must also be smaller than a certain threshold (ideally of size 1).

As shown in Table 6.1, we find that both algorithms have similar success rates: The naive algorithm manages to identify the true origin in $61\% \pm 3\%$[5] cases, while the graph-based traceback algorithm achieves a slightly higher success rate of $68\% \pm 3\%$ with a candidate set of 8 or less. When limiting the graph-based algorithm to a single candidate, it still manages to succeed in $58\% \pm 3\%$ cases. The naive algorithm is expected to find all on-path ASes and thus has an expected candidate set size of the average path length. For the graph algorithm, we will use a candidate set size of 8 in the following.

6.8.2.2 Runtime

As noted before, the traceback speed of BGPEEK-A-BOO is limited by the rate at which BGP advertisements can be sent. We therefore measure the runtime of each algorithm in the number of probing steps, with a realistic step duration of 10 minutes. As can be seen from Figure 6.3, the naive algorithm performs similar to the number of steps derived in Section 6.6.1, with a mean and median runtime of 549 steps. Adding early termination on stub-ASes slightly reduces the median to 523 steps, but more importantly reduces the average to just above 400 steps, with 25% even terminating in under 268 steps. Since it is strictly superior to the naive algorithm, we will only report results for the naive+ algorithm in the following.

Overall, the graph-based algorithm performs best by far, with an average of 159 steps and a median of only 98.5 steps. Interestingly, a quarter of all cases terminate in less than 29 steps, around 4.83 hours at six advertisements per hour. Most notably, even in the worst

[5]We denote the 95% Agresti-Coull confidence interval

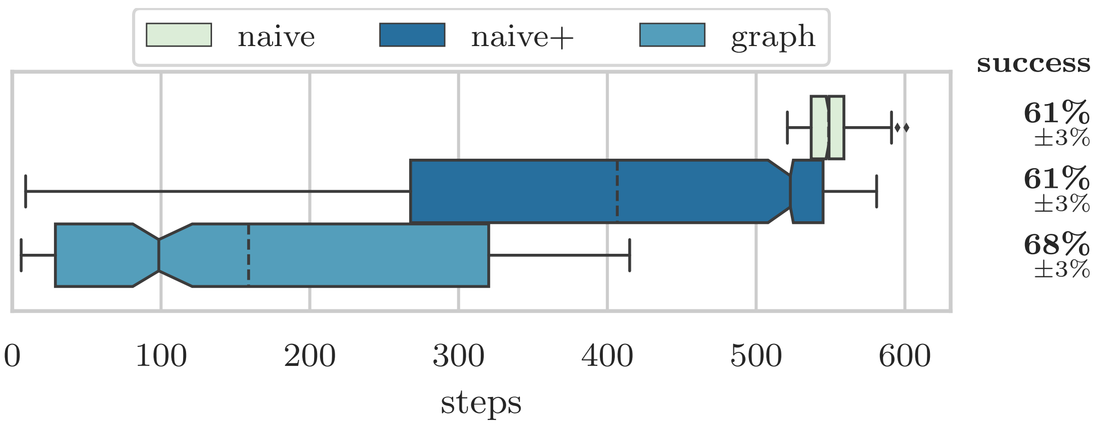


Figure 6.3: Runtime comparison. Dashed lines show average times, while the notch indicates the 95% confidence interval around the median.

case the graph-based algorithm still completes its search in 415 steps or less, thus making it faster than even the best run of the naive algorithm. This demonstrates that utilizing AS graph information substantially improves the efficiency, with a median runtime speed-up of 5.6×.

6.8.2.3 Prefix Parallelization

If BGPEEK-A-BOO observes an attack in multiple probing prefixes simultaneously, we can further speed up traceback by running multiple probes in parallel. While this does not necessarily reduce the total number of *advertisements*, it greatly reduces the number of *steps* and hence the total runtime. Figure 6.4 therefore compares using different numbers of probing prefixes, one (no parallelization), two, and eight. For the naive algorithm we see a linear speed-up, reducing the median runtime from 523 to 262 steps when using two prefixes and 66 steps when using eight. With the exception of the "binary search" step, probe selection is independent from previous probe results, and thus the naive algorithm can be almost perfectly parallelized. The graph-based algorithm benefits less from parallelization, since probe selection is strongly coupled to previous probe results, but still sees a 5.8× speed-up in the median runtime from 98.5 steps to 17 steps when going to eight prefixes. As with the probe size, varying the number of prefixes does not have a significant influence on the success rate.

6.8.2.4 Default Routes

Intuitively, default routes should have a negative influence on our traceback approach, both in terms of runtime and success. As the presence of default routes has been confirmed by multiple studies [20, 134] we would like to quantify to which extent they impact our results. To this end, we repeated the experiment two more times, once in an “ideal world” setting with no default routes and once in an exaggerated setting where 80% of ASes have a default route. In line with intuition, both algorithms generally perform better with fewer default routes, as shown in Figure 6.5. Interestingly though, the graph-based algorithm

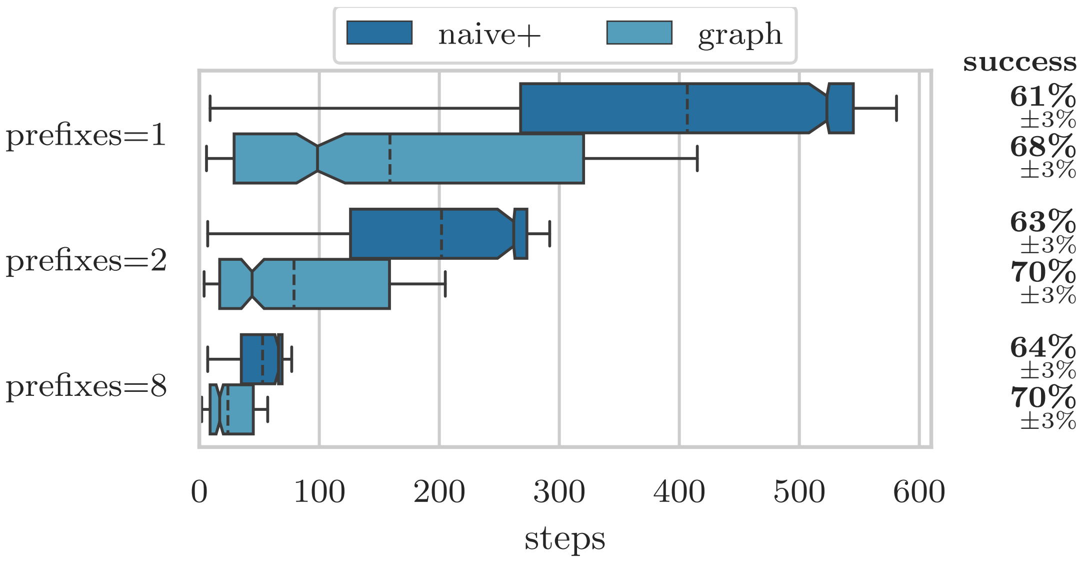


Figure 6.4: Influence of parallelization

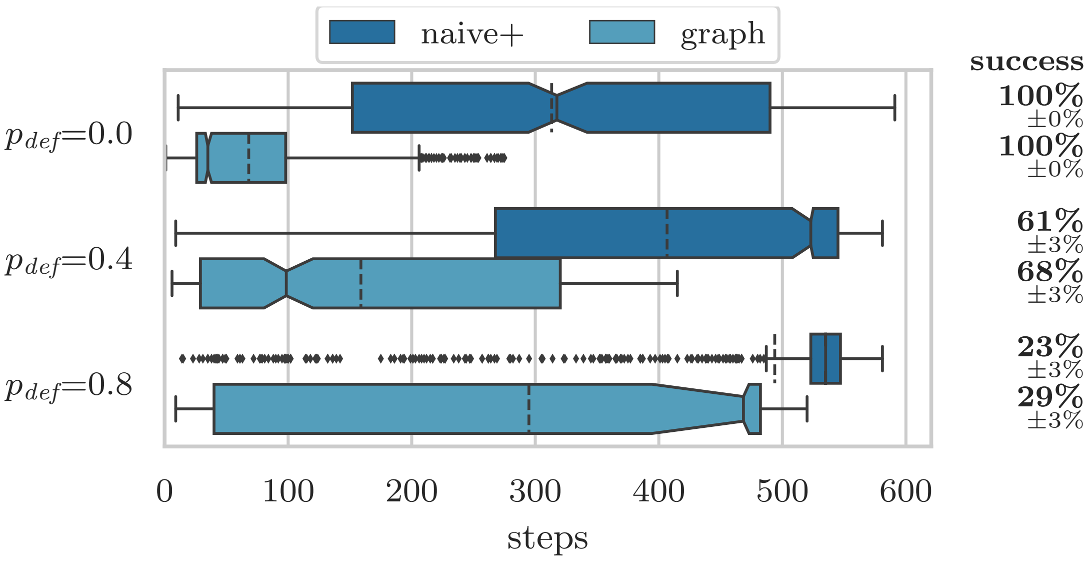


Figure 6.5: Influence of default route prevalence

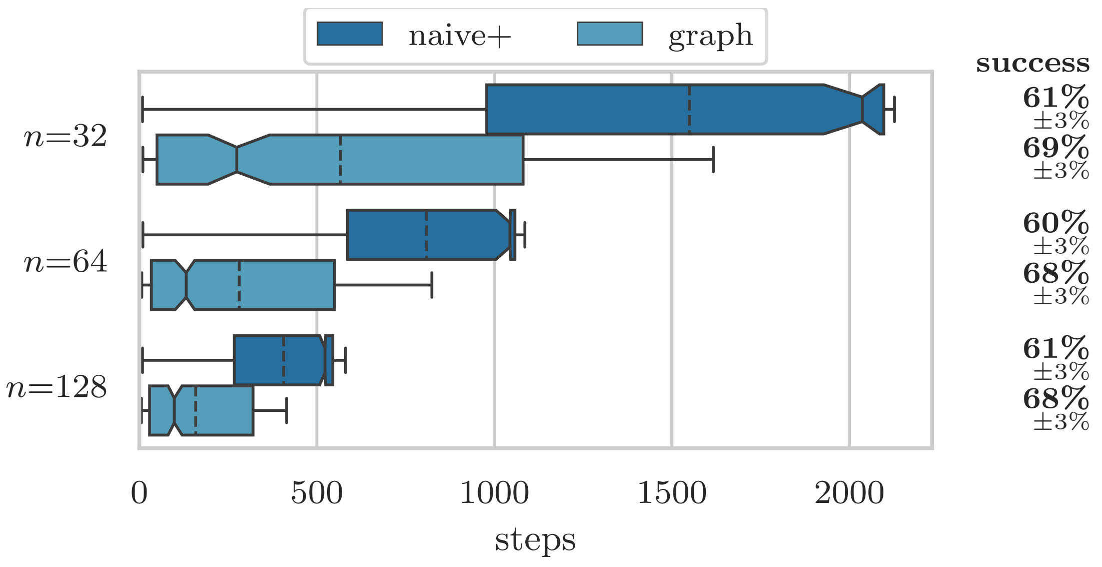


Figure 6.6: Influence of probe size

still terminates faster when faced with 40% default routes than the naive algorithm in the ideal world setting without any default routes. The default route prevalence is also the most determining factor of traceback success: In the idealized setting with no default routes, all simulated runs were successful, resulting in an estimated success rate of 100% ± 0%. Yet, even with 80% default routes, they achieve a success rate of 23% ± 3% and 29% ± 3% respectively.

6.8.2.5 Probe Size

We ran another experiment to measure the influence of the probe size, evaluating each algorithm with a probe size of $n = 32$, $n = 64$, and $n = 128$. The results, shown in Figure 6.6 confirm our intuition that the runtime of the naive algorithm scales inversely to the probe size. Where at $n = 128$ the naive algorithm has a median runtime of around 500 steps, this doubles to just over 1 000 at $n = 64$ and quadruples to 2 000 at $n = 32$. While the same relation holds true for the graph algorithm's *maximum* runtime, 400 steps at $n = 128$ to 1 600 steps at $n = 32$, it still achieves a median runtime of less than 500 steps even with $n = 32$. At that, it outperforms both naive variants with a probe size of $n = 128$. As expected, the success rate of BGPeek-a-Boo is not influenced by the probe size.

6.8.2.6 Deployment Location

To quantify the impact of the deployment location of BGPeek-a-Boo, we also ran the experiment from three different ASes: Next to a deployment at a PEERING customer we also simulated runs from a Tier-1 provider (AS174) as well as from a national research network. However, as shown in Figure 6.7, neither runtime nor success rate vary significantly between the three different deployments.

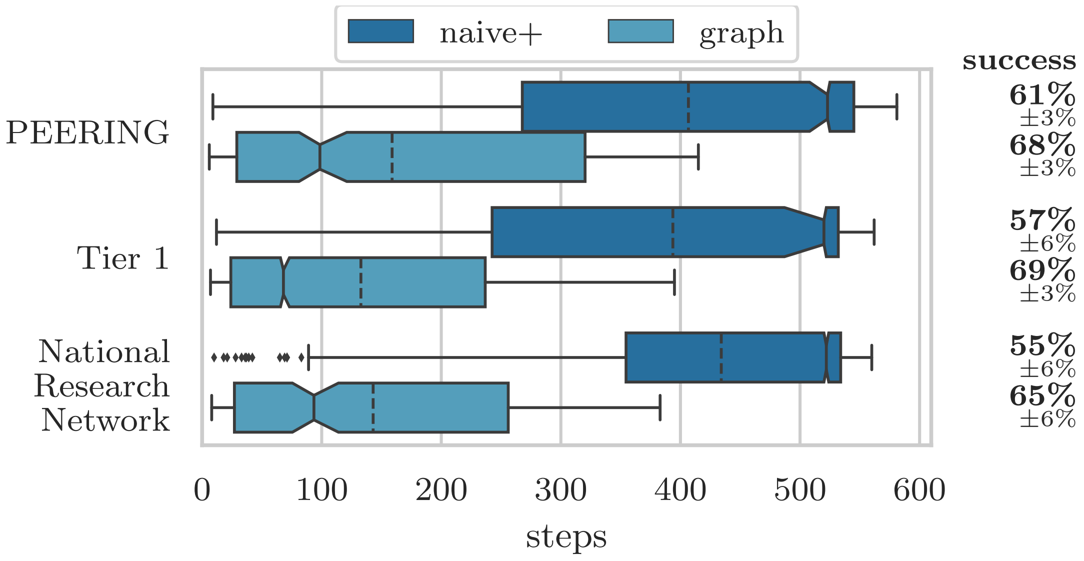


Figure 6.7: Influence of deployment location

6.8.2.7 TTL Values

BGPEEK-A-BOO relies on TTL values to detect when traffic is redirected to alternative routes. However, while mostly stable, TTL values can change for reasons unrelated to our traceback and could even be modified by an attacker attempting to evade detection. In a final experiment we therefore simulate how BGPEEK-A-BOO performs *without* TTL values, i.e., only checking whether poisoning leads to a stop in traffic. Perhaps surprisingly, the results in Figure 6.8 show that a lack of TTL values only leads to a slowdown of the graph algorithm, whose median runtime doubles, but has no statistically significant impact on overall traceback success. This shows that BGPEEK-A-BOO can still be used even if TTL values are found to be unreliable, albeit with increased runtime.

6.8.3 Supporting Real-World Experiments

We also leveraged the PEERING BGP testbed [128] to conduct experiments in the live Internet and analyzed attack data captured by AMPPOT [S1], to bootstrap additionally required parameters and to assess the plausibility of our simulated results.

6.8.3.1 Changing Default Routes

To faithfully model the effect of default routes, we not only need to know how many ASes *have* a default route, but also *how often* this default route differs from the regular route. To measure this effect, we designed the following experiment: From our temporary AS we would send out a poisoning advertisement for a target system, advertising one of three /24 prefixes from our /22 allocation. The fourth prefix would be advertised regularly to serve as a control. After a short while we would then ping the target system from both, the poisoning prefix and the control prefix. If ping replies are still observed in the poisoning prefix we can conclude that the target does have a default route. If the TTL values also differ between the

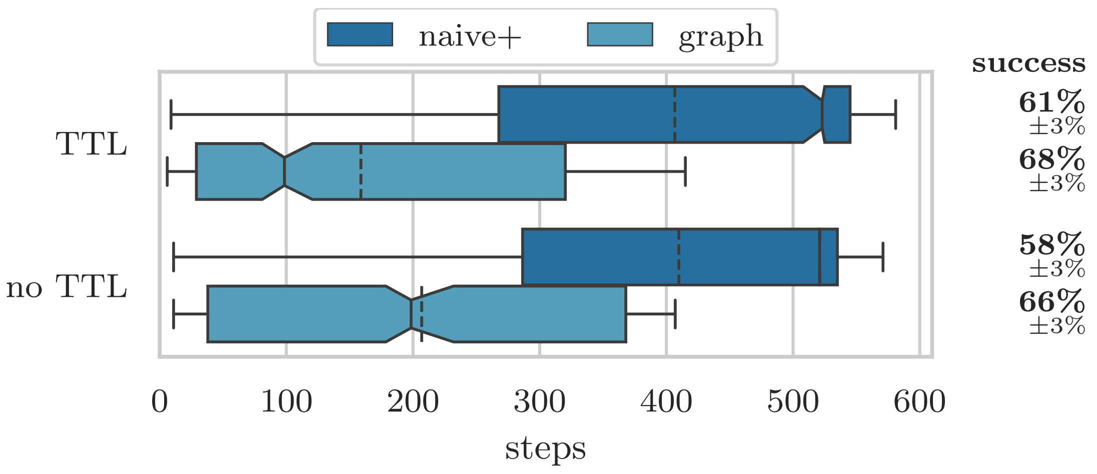


Figure 6.8: Influence of using TTL values

poisoning and the control prefix we can further infer that it differs from the regular route.

As targets we randomly selected 624 RIPE Atlas probes [117] in different ASes. TTL values were recorded five minutes after the advertisement was sent to allow routes to settle [82, 74], and new advertisements were only sent every ten minutes per prefix. Out of the 624 tested RIPE Atlas probes we found 360 (58%) to have default routes, i.e., we would still receive pings for the prefix that was advertised with a poisoning advertisement only, a fraction larger than the default route prevalence reported by Smith et al. [134] in 2020, but lower than the one reported by Bush et al. [20] in 2009. We attribute the discrepancy to two effects: First, our sample size is smaller than the one employed by both Smith et al. and Bush et al. and further biased towards ASes housing RIPE Atlas probes. Second, we found the probe's AS information as reported by RIPE's Atlas back end to not always be accurate and thus suspect that in some cases the probe's *actual* AS was different from the one poisoned, thereby making them a false positive. As noted in Section 6.8.2, we picked a default route probability of 40% for our simulator.

In 101 of 360 (28%) cases we further observed a discrepancy in TTL values between the poisoned and the control prefix, letting us conclude that in these cases the default route in fact differs from the regular route. At a confidence level of 95% this thus gives a probability of having a differing default route between 23% and 33%. We therefore model this in our simulator by having an AS choose a different upstream in 30% of the cases when poisoned.

6.8.3.2 On-Path Poisoning

As a main primitive our approach relies on the assumption that poisoning on-path ASes provokes some observable change in the target traffic, either a change in TTLs due to a route change or a complete absence of traffic due to the lack of alternative routes. To measure how this assumption holds up in practice we designed the following experiment: As the "traffic origin" we randomly selected a RIPE Atlas probe [117], and used a stream of ping packets from the probe to our traceback system deployed at PEERING as the "attack traffic". To obtain a real-time approximation of the taken AS path, we scheduled a traceroute measurement from the Atlas probe to our traceback system. Running the traceroute in that direction ensures that we measure the actual ingress path to our system and do not have to assume paths to be symmetric. Using the Team Cymru IP to ASN Lookup [149] we then mapped traceroute IP hops to ASes and, subsequently, poisoned every discovered on-path AS one-by-one. In contrast to the previous experiment, this time we did not measure the impact on the poisoned AS, but on the ping packets from the Atlas probe simulating the target traffic flow. As before we waited five minutes before taking measurements after new advertisements and ten minutes between advertisements for the same prefix.

Note that the list of on-path ASes obtained that way is naturally incomplete, as a traceroute may miss hops along the path and the IP-to-ASN mapping leads to further inaccuracies as well [170, 66]. To validate the real-time IP-to-ASN mapping we obtained from Team Cymru we later compared the mapping results to the data covering the same timeframe published on RIPEstat [119]. Here we found 22 instances in which the IP-to-ASN mappings disagreed. Manually analyzing these 22 cases we determined the Team Cymru mapping to be correct in 14 of these, and excluded the other 8 from further analysis. As we were only interested in measuring true on-path ASes we further also excluded hops that mapped to the same AS as the Atlas probe mimicking the traffic source.

We ran the experiment with Atlas probes located in 161 different ASes, allowing us to collect a total of 327 unique (on-path AS, target AS)-pairs. In total, we found that poisoning an on-path AS resulted in a loss of traffic in 137 (42%) cases and a change in the TTL in further 112 (34%) cases. Only in 78 (24%) instances we found poisoning on-path ASes to have no measurable impact on the target traffic at all.

To further analyze whether the impact is related on the distance between the poisoned and the measured AS, we grouped the results by their hop distance to the target AS. While ideally we would have used *AS* hop distances for this, this would have required access to the full AS path. We thus resorted to using *IP* hop distances as they could be obtained easily from the traceroute, and let the poisoning results count towards all IP hops that were mapped to the same AS. Figure 6.9 shows the normalized results per IP hop count, with the black line denoting the number of tested AS pairs per hop distance to give an indication of their significance. As depicted, most on-path ASes were discovered at a distance of 2 to 10 hops from the origin. However, we find that overall the distance between the poisoned AS and the traffic source appears to have little impact on how traffic is affected, and that in most cases a measurable impact can be expected.

6.8.3.3 TTL Distribution

We leveraged the same RIPE Atlas traceroutes to obtain a realistic model of inner-AS path lengths for our simulator, such that we can simulate IP hop counts of AS paths. For this, we utilized the same IP-to-AS mapping and then scanned the traceroute results for consecutive hops in different ASes. If we can find two of these transitions, $A \rightarrow B$ at hops $(x, x+1)$ and $B \rightarrow C$ at hops $(y, y+1)$, then the inner-AS path length of B from A to C is $y+1-x$ hops. We were able to extract 137 AS-triples and their corresponding hop counts and found that a negative binomial distribution with $k=3, p=0.62$ was a good fit.

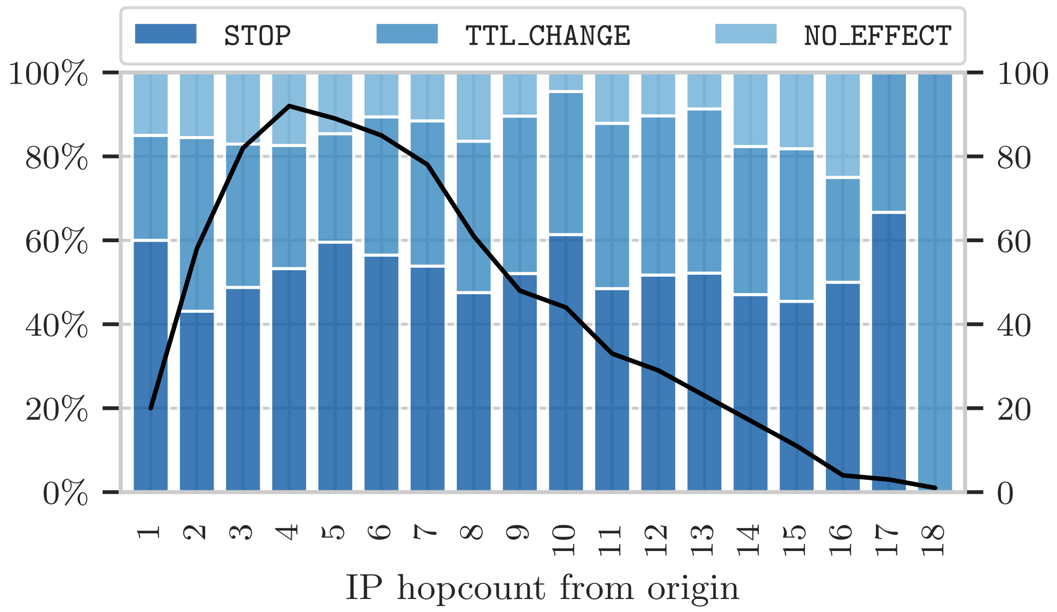


Figure 6.9: Real-world results of on-path AS poisoning

6.8.3.4 Continuity of Spoofing Activity

Even with the graph-based algorithm, BGPEEK-A-BOO has an average runtime of 147 steps, or just over one day at six advertisements per hour. We therefore assess, how long an attack source may be observed consecutively. To this end, we collected data on 13 321 740 amplification attacks observed by AMPPOT [S1]. All attacks were collected between 2015-11-25 and 2020-06-15 by a *Selective Response* enabled honeypot [P1]. As discussed in Section 4.6, Selective Response restricts every scanner to finding a different set of 24 of the 48 honeypot IPs, thereby imposing a unique fingerprint on the scanner. In the previous chapter we have seen that these fingerprints can be used to re-identify booter services, who scan for amplifiers at semi-regular intervals but use a stable amplifier set between scans. Assuming that a similar behavior holds true also for other attack sources and that scan results are not shared amongst multiple attackers, we can use the fingerprint as an identifier for the (unknown) traffic source.

To focus on distinctive fingerprints, we only considered attacks that used at least 12 and at most 24 honeypot IPs, which left us with 8 635 257 remaining attacks. For each fingerprint, we then computed the longest period, during which it was observed at least 90% (99%) of the time. Figure 6.10 shows the fraction of *attacks* whose fingerprint was observed for a given duration at a given activity level. From this plot we find that the majority of attacks stem from sources that are also active for extended periods of time. For example, over 68% of attacks stem from a source that was seen for over a week at 90% activity, 51% even at 99%.

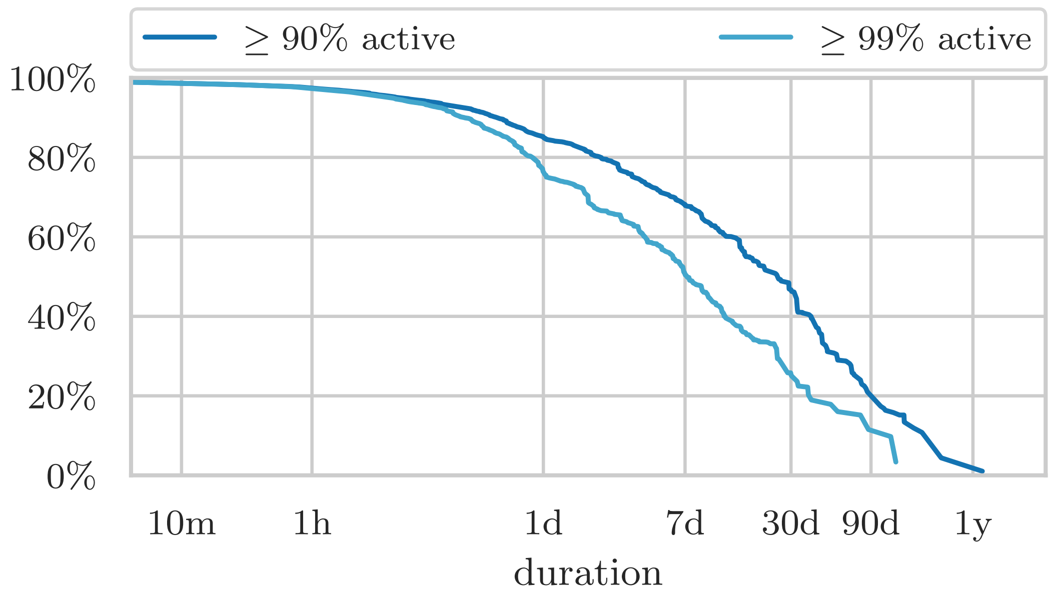


Figure 6.10: Fingerprint activity

6.9 Discussion

We now discuss how our evaluation results translate to the real-world deployability of our approach and the ethics of our live Internet experiments.

6.9.1 BGP Mechanisms

While our simulator strives to faithfully model routes and advertisement propagation, it does so by abstracting ASes and their relationships into a graph model. This abstraction may not fully model BGP in the real world, where ASes do not act as atomic entities but advertisements are instead passed between routers and processed by actual BGP implementations. As such, multiple mechanisms may influence the propagation of advertisements which we discuss below.

6.9.1.1 Route Flap Dampening

Route Flap Dampening (RFD) [157] aims to decrease the load on routers by maintaining a per-route penalty score. This score is increased for every update, but decays exponentially over time. Once a route's score reaches a threshold, it is no longer considered for routing nor propagated to peers until its score drops below a re-use threshold. Studies have shown that the default thresholds used in RFD can actually be harmful even for stable routes [92], and RFD was subsequently advised against [136]. Although later studies proposed new settings that have less adverse effects [21, 105, 106], it still remains disabled by default in, e.g., Cisco routers [28]. However, as RFD maintains a score on a *per-route* rather than a *per-prefix* basis, it does not affect our traceback technique: Whilst we send out multiple updates for the same prefix, every update includes a new AS path and therefore constitutes a new route [157, sec. 4.4.3]. Hence, even if RFD was enabled, our traceback should still work.

6.9.1.2 Minimum Route Advertisement Interval

Another measure to prevent high load on BGP routers is the *Minimum Route Advertisement Interval* (MRAI), that limits the rate at which updates for a certain prefix are passed on to peers. The idea behind this is that withholding routes for a certain time allows the (downstream) path exploration to converge, thereby reducing the number of updates and withdrawals sent further to peers. The recommended value for the MRAI timer is 30 seconds [115, sec. 10]. As we always waited ten minutes between advertisements this should have given routers ample time for this timer to expire. However, it also means that a real-world deployment of our approaches should employ a similar delay between advertisements.

6.9.1.3 Path Filtering

Smith et al. also report ASes to filter advertisements based on path lengths or by checking for potential poisoning paths [134]. For length-based filters Section 6.8.2.5 indicates that our graph-based approach would still perform well in many cases even when limiting the path length to 32. However, filtering of poisoning advertisements can impede our traceback approach if it is performed by one of the on-path ASes. In this case, *any* poisoning advertisement may provoke the target traffic to change and thus both algorithms may falsely flag the probed AS to be on-path—even if the observed change was only caused by filtering.

6.9.1.4 Non-Uniform Routing

The use of active probing in our traceback assumes that all outbound packets from an AS are affected in the same way by a route change, regardless of whether they are originating from the AS or forwarded on behalf of another. In theory, every edge-router of an AS could behave differently and use a different route, which in turns means that different hosts in the AS could behave differently under poisoned advertisements. A study by Mühlbauer et al. [98] finds that such route diversity can be modeled by splitting ASes into multiple *quasi-routers*, each modeling a consistent routing behavior observed by the AS. However, they find that the vast majority in route diversity comes from prefix-dependent preferences, and that “for almost all ASes one quasi-router suffices”. Since in our case the attack traffic as well as the active measurement replies are destined towards the *same prefix*, they are not affected differently by prefix-dependent routing preferences. We can thus conclude that, for our purposes, *all* outbound traffic from an AS towards our prefix takes the same AS level path.

6.9.2 Observation Correlation

Our approach relies on stopping traffic and changing TTL values to infer AS-level route changes. However, TTL values along a path may also change for other reasons (e.g., inner-AS route changes), and attack traffic may cease because the attack stopped entirely. Therefore, if it is unclear whether a change was the result of a poisoning advertisement, the advertisement can be repeatedly withdrawn and re-advertised until a correlation can be confirmed or refuted. Furthermore, if BGPEEK-A-BOO's control honeypots also observe the same attack traffic, they too can be used to decide whether a change is spurious.

6.9.3 AS Flow Graph Correctness

Whereas our naive algorithm makes no assumptions about AS relationships, our graph-based traceback algorithm assumes that

1. AS paths adhere to the valley-free assumption and AS relationships follow the standard customer-provider/peer-to-peer model, and
2. a global view of these relations is available.

We discuss both assumptions in detail below.

6.9.3.1 Valley-Free Assumption and AS Relationships

While both customer-provider and peer-to-peer relations between ASes are well-established and have intuitive economic incentives, the exact relation between two ASes can be arbitrarily complex. For example, Giotsas et al. [59] identified 4 026 ASes whose relationships they classified as *hybrid* or *partial transit*. In a hybrid relation, two ASes exhibit different relations at different exchanges, whereas a partial transit relation is a restricted form of a customer-provider relation. Giotsas and Zhou [60] also find a small number of AS paths that seemingly violate the valley-free assumption, which they attribute to non-standard AS relationships.

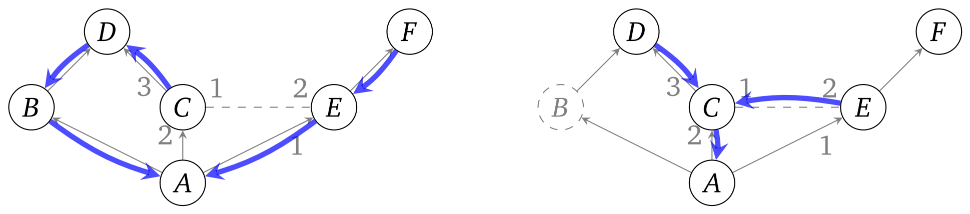
Our AS flow graph only captures customer-provider and peer-to-peer relations and requires paths to adhere to the valley-free assumption. Thus, our graph-based traceback approach may falsely exclude ASes it believes to be reachable or dominated by others when faced with such non-standard relations. However, as long as reachability and dominance information can be efficiently encoded (e.g., through adapting links in the graph), a similar algorithm may still be employed.

6.9.3.2 AS Relationship Dataset

AS relationships are usually subject to non-disclosure agreement and can thus only be inferred from publicly available routing data. While the state-of-the-art of inferring the global AS relationship graph has been constantly evolving [53, 144, 38, 46, 163, 40, 39, 65, 90], inference will inevitably only be able to produce an approximation of the AS graph. As with non-standard AS relations, missing or incorrect links are problematic for our graph-based algorithm, which could cause it to wrongly discard ASes as candidates. In that regard, our simulation results should be seen as best-case results, as both simulator and traceback use the same graph data.

6.9.4 Induced Route Changes

In some cases, a poisoned advertisement can lead to route changes or losses even at ASes that can never receive an advertisement through one of the poisoned ASes. Consider for example



(a) A is customer of B, C, and E; B and C are customers of D; E is customer of F; C and E have a peer-to-peer relation; numbers indicate local preferences; thick arrows show the resulting routes to A.

(b) Poisoning B causes C to switch routes, which induces a route change at E and a loss of connection at F.

Figure 6.11: Example of induced changes

the network shown in Figure 6.11. In the normal state, AS C receives three advertisements for routes to A, the direct route from A, the route $D \rightarrow B \rightarrow A$ from D, and the route $E \rightarrow A$ from E. From these it will choose the route via D, since it has the highest local preference. However, since this route is received from one of C's providers, it cannot be exported to the peer E. AS E therefore only sees the direct route from A, which it can further advertise to its provider F.

Poisoning B makes the route $D \rightarrow B \rightarrow A$ unavailable. C therefore switches to its next preferred route, the direct route to A. Since A is a customer of C, C can advertise this new route $C \rightarrow A$ to its peer E. This, however, induces a route change at E, because this new route has a higher local preference at E. Furthermore, since the best route at E is now received from a peer, it can no longer be exported to E's provider F. F therefore loses its connection to A. Note that neither E nor F could ever have a route to A via the poisoned AS B. Yet, they see a route change or even a loss of connectivity.

An AS can only cause such *induced* changes, if it can receive two different routes, one from a customer and one from a peer or provider. Only in that case, the set of other ASes that it can export routes to can change: Switching from a customer-provided route to a peer/provider-provided one limits it to advertise this route to its customers, switching the other way enables it to also advertise a route to its peers and providers. We will call these ASes *ambiguous*. ASes that are reachable through an ambiguous AS may therefore experience induced changes. Further, if an AS is *only* reachable through peers or providers of ambiguous ASes, it may also experience an induced connectivity loss.

For our naive algorithm, such induced changes can cause additional ASes to appear in the final result set (e.g., in the example above, B would be erroneously considered *on-path* even if the attack came from F). Yet, these induced changes cannot "hide" actual on-path ASes from detection. Our graph algorithm on the other hands requires a small modification (shown in Algorithm 6.3) to correctly handle induced changes: Whenever the candidate set is reduced, ASes that could show the observed behavior due to induced changes need to be retained.

To assess the impact of induced route changes, we ran a simulation of our graph-based traceback with these modifications. As shown in Figure 6.12, neither runtime nor success

Algorithm 6.3 Flow Graph based traceback algorithm adapted to handle induced route changes

```
procedure UpdatePassive'($\mathscr{P}$, $r_{\text{passive}}$)
    if $r_{\text{passive}} = \texttt{STOP}$ then
        $\mathscr{C}_{stop} \leftarrow \text{indStop}_{G_A}(\mathscr{P})$
        $\mathscr{C} \leftarrow \mathscr{C} \cap \left(\text{dominatees}_{G_A}(\mathscr{P}) \cup \mathscr{C}_{stop}\right)$
    else if $r_{\text{passive}} = \texttt{TTL\_CHANGE}$ then
        $\mathscr{C}_{change} \leftarrow \text{indChange}_{G_A}(\mathscr{P})$
        $\mathscr{C} \leftarrow \mathscr{C} \cap \left(\text{reachable}_{G_A}(\mathscr{P}) \cup \mathscr{C}_{change}\right)$

procedure UpdateActive'($\mathscr{P}$, $r_{\text{passive}}$, $\vec{r}_{\text{active}}$)
    if $r_{\text{passive}} = \texttt{STOP}$ then
        $\mathscr{P}_{\text{inconsistent}} \leftarrow \{X \in \mathscr{P} \mid \vec{r}_{\text{active}}[X] \neq \texttt{STOP}\}$
        $\mathscr{C}_{stop} \leftarrow \text{indStop}_{G_A}(\mathscr{P})$
        $\mathscr{C} \leftarrow \mathscr{C} \setminus \left(\text{reachable}\, G_A(\mathscr{P}_{\text{inconsistent}}) \setminus \mathscr{C}_{stop}\right)$
    else
        $\mathscr{P}_{\text{inconsistent}} \leftarrow \{X \in \mathscr{P} \mid \vec{r}_{\text{active}}[X] = \texttt{STOP}\}$
        $\mathscr{C}_{change} \leftarrow \text{indChange}_{G_A}(\mathscr{P})$
        $\mathscr{C} \leftarrow \mathscr{C} \setminus \left(\text{dominatees}_{G_A}(\mathscr{P}_{\text{inconsistent}}) \setminus \mathscr{C}_{change}\right)$
    $\mathscr{P} \leftarrow \mathscr{P} \setminus \mathscr{P}_{\text{inconsistent}}$
```

rate differ significantly compared to the unmodified version. This can be explained because ambiguous ASes are relatively rare: Analyzing the flowgraph used during simulation reveals only 231 ambiguous ASes. We can thus conclude, that induced route changes only have negligible impact on the traceback performance.

6.9.5 Multi-Source Attacks

As also discussed in Section 5.6.1, amplification attacks are largely launched from single sources [S1, P2]. For BGPeek-a-Boo we thus assume that every attack has a unique source AS. However, amplification attacks could theoretically also be launched from multiple colluding sources in different ASes. Fortunately, as long as the attack traffic of a multi-source attack can be separated by source (e.g., by different TTL values due to different path lengths), both algorithms of BGPeek-a-Boo still work as before. Otherwise, successful poisoning of one of the sources can only be measured as a decrease in attack traffic volume. Yet, even in that case, the naive algorithm should be able to find all attack sources.

6.9.6 Evasion

During a traceback run BGPeek-a-Boo creates a large number of route advertisements for the probing prefixes. Attackers that are aware of our system may therefore try to evade it by monitoring public BGP data [156, 118] in order to identify and exclude the probing prefixes. While we cannot *prevent* such active evasion attempts, we can at least *detect* if an attacker is evading our system. In this case, we would not observe any attacks from a given

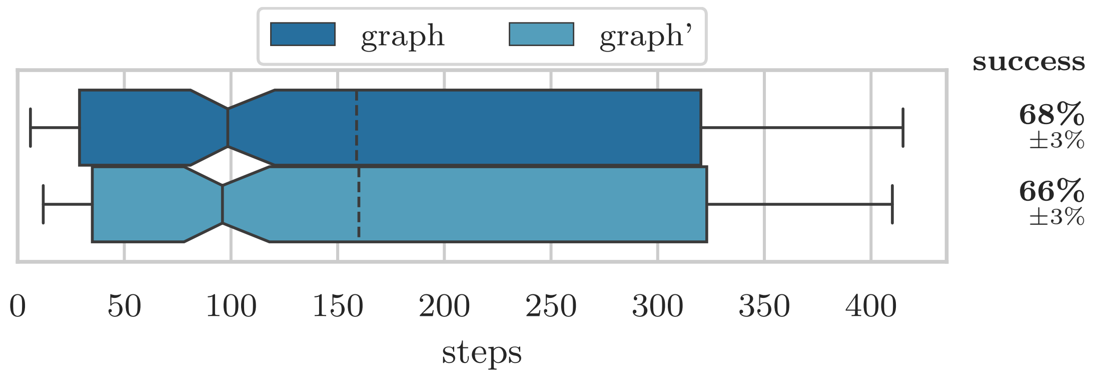


Figure 6.12: Influence of induced route changes. The modified algorithm is labeled *graph'*.

adversary at the probe honeypots, but would keep observing them at the control honeypots. Yet, even outside of BGPEEK-A-BOO, amplification honeypots are inherently detectable due to their rate limiting behavior [S1].

Besides evasion, attackers might also attempt to deceive BGPEEK-A-BOO by modifying their initial TTL values. However, as shown in Section 6.8.2.7, BGPEEK-A-BOO still performs well even when ignoring TTL values.

6.9.7 Ethical Considerations

We took several measures to ensure that our experiments did not impact other systems or lead to instabilities in the BGP. To obtain a real-time estimate of the currently active BGP path, we conducted traceroute runs from RIPE Atlas probes to our measurement system hosted at PEERING. In order to keep the impact on other systems minimal, we only used one-off measurements with the default values defines by RIPE (i.e., packets of 48 bytes, at most 3 packets per destination). For our active ping measurements we ensured to send at most one packet per minute per target on average. At 64 bytes per packet (1Bps), we believe these to have negligible impact.

All experiments that involved sending (poisoned) BGP advertisements were conducted only after consulting PEERING operators. To further ensure that other experiments running at the testbed were not influenced by ours, we used temporary prefixes and a temporary ASN allocated by our regional Internet registry for the purpose of these experiments. We also registered our temporary allocations in the WHOIS database, such that network operators were able to contact us directly. Additionally, we closely monitored network operator mailing lists.

6.9.7.1 Impact on Legitimate Traffic

Poisoning advertisements can render a prefix temporarily unreachable from parts of the Internet. Therefore, BGPEEK-A-BOO's probing prefixes should host no other systems but honeypot reflectors. As we use BGP Poisoning on these small prefixes only, other prefixes remain unaffected—effectively excluding collateral damage on benign traffic.

6.10 Related Work

We find related work from two fields of study: Next to related work studying the problem of IP spoofing and traceback, which we discuss in Chapter 3, BGP Poisoning is also studied as a general primitive for traffic engineering and security applications.

Closely related to BGPEEK-A-BOO is the concurrent work of Fonseca et al. [50], which attempts to identify spoofing sources by varying BGP advertisements from multiple anycast locations. For this, they create 700 different advertisement configurations by selecting subsets of their peers, adding path prepends, and poisoning immediate neighbors. For every configuration they then record through which peer the attack traffic is received, thereby generating a fingerprint of source ASes. In contrast to BGPEEK-A-BOO, their approach necessarily requires 700 probing steps and cannot distinguish ASes that share a common path beyond their immediate neighbors.

6.10.1 BGP Poisoning

Although only a side-effect of BGP's loop detection mechanism, BGP Poisoning has seen a wide field of applications. In the area of measurement studies, Colitti et al. [32] and Anwar et al. [5] employed BGP Poisoning to supplement the analysis of AS relationships and prefix propagation. Katz-Bassett et al. [73, 74] showed how BGP Poisoning may be used to actively repair routes, and Smith et al. [135] later showed how it can be used to avoid DDoS congested links. Finally, works by Tran et al. [155] and Smith et al. [134] analyze how well BGP Poisoning works in practice through extensive measurements. Yet, to the best of our knowledge, our approach and the concurrent work by Fonseca et al. [50] are the first to leverage BGP Poisoning for DDoS attack traceback.

6.11 Conclusion

IP spoofing not only enables amplification attacks, but also hides the attackers' true whereabouts. BGPEEK-A-BOO employs a novel traceback approach that shows that BGP Poisoning can be used to track down an attacker's network location—requiring neither the assistance of external parties nor knowing the attacker in advance. We find that our naive algorithm has a median runtime of 549 steps, or just under four days with realistic parameters, thus showing the feasibility of our approach in practice. Our second algorithm leverages a graph model of BGP path propagation built from AS relationship data and manages to reduce this runtime to 98.5 steps, or just over one day, for the same parameters—and in only 29 steps, under five hours, in a quarter of all cases.

7
AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities (and Using Those to Synthesize DDoS Honeypots)

7.1 Motivation

In the previous chapters we discussed traceback techniques to find the perpetrators behind amplification attacks. While attack attribution provides an important step towards effective criminal prosecution of attackers, it ultimately remains a reactive measure. An equally important step in the fight against amplification DDoS attacks is thus the proactive identification and mitigation of amplification attack vectors, which enable these attacks in the first place. However, all amplification attack vectors known to date were either found by researchers through laborious manual analysis or could only be identified *post-postmortem* following large attacks.

To this end, we present AmpFuzz, the first systematic approach to finding amplification vectors in UDP services. AmpFuzz is based on guided greybox fuzzing boosted by a novel static analysis technique to make fuzzing UDP-aware, which gives up to a thousandfold increase in performance. We evaluate AmpFuzz on 25 Debian network services, where we (re-)discover 5 known and 4 previously unreported amplification vulnerabilities.

Finally, we leverage a modified symbolic execution engine to automatically synthesize amplification honeypots, which allow to monitor malicious scan and DDoS attack activity and also serve as a core component for our traceback mechanisms.

7.2 Problem Description

Every time an amplification vector is discovered, we observe record-breaking attacks abusing the new vulnerability. For example, only shortly after an amplification vector was found in Memcached (a distributed memory-caching system), attackers abused its gigantic potential [31]. Similarly, the discovery of amplification vectors in RIPv1 [64] has quickly led to a stark increase in amplification abuses of this protocol [19]. Surprisingly, despite this history of documented amplification vulnerabilities, we lack any automation in *discovering* harmful protocol implementations. To date, amplification vulnerability search is a largely manual effort, typically driven by attacker groups in search for unknown and thus unfiltered amplification vectors for which no mitigation strategies exist. Worse, new vulnerabilities quickly gain wide popularity among fellow attackers. Consequently, defenders lag behind and mostly react. Any future amplification vulnerability will over and over trigger gigantic DDoS patterns for which network operators and anti-DDoS services are not well prepared.

There are strong incentives to automate vulnerability search. First, early discovery allows to safeguard protocols and their implementations against known amplification vectors as early as possible—ideally before their abuse. There are several success stories in which implementation changes and/or large-scale disclosure operations have massively reduced the number of vulnerable services [80, 158]. Second, early knowledge of vulnerabilities allows to monitor amplification attacks with the help of amplification DDoS honeypots [S1, 153]. Honeypots not only provide an overview of the situation and can alert victims in real time. In fact, they form the foundation for mechanisms which trace back amplification attacks to their origin that are actively used by law enforcement agencies [P1, P2, P3, 50]. Yet, honeypots are “blind” unless they know which vulnerabilities to emulate and monitor. Third, anti-DDoS services can ingest novel attack vectors in their automated defense systems that filter attack traffic *prior to* attack abuse. This would be a game-changer to the reactive defensive situation and give defenders sufficient heads-up to create proactive defense strategies.

Unfortunately, although there is rich literature on discovering other types of software vulnerabilities, amplification represents a bug class on its own. In fact, searching for amplification vulnerabilities raises several challenges. First and foremost, there is a plethora of network protocols and implementations thereof, for only few of which there exists a formal specification on protocol states and message formats. In other words, we deal with unknown protocols for which we cannot assume a priori knowledge. The UDP stack brings additional challenges, as one has to decide (i) when a service is in a state in which it can react to requests, and (ii) when a service has finished processing a request. Finally, the type of vulnerability is vastly different from other bug classes that can be detected using simple binary indicators, e.g., by observing program crashes or inconsistent program states.

7.3 Contributions

In this chapter, we provide the first principled approach to find amplification vulnerabilities in network services. We propose a greybox fuzzer which aims to discover program inputs over the network that result in significantly larger responses. We overcome the aforementioned challenges by using a directed fuzzer which prioritizes paths leading to sending functions (e.g., `sendto`) which we extend with UDP-awareness. To this end, we use a combination of static analysis and lightweight instrumentation to notify the fuzzer when input is expected by the program. These techniques also enable us to proactively terminate fuzzing executions that generate no output instead of having to rely on expensive request timeouts. Finally, a simple yet effective mutation strategy maximizes the amplification factor of potential vulnerabilities.

We then leverage the fact that vulnerability *detection* can provide the necessary means to *monitor* amplification abuse in an automated way. Until now, amplification DDoS honeypots solely rely on experts who have to craft protocol feature-specific replies to attacker probes by hand. We expand our methodology to synthesize request handler routines for the identified vulnerabilities which can be plugged into amplification honeypots. We automate this process by leveraging symbolic execution to generate path constraints and output expressions for each discovered vulnerability. These can then be used to match probing requests received by the honeypot, and to select replies that attackers find attractive. We can thereby completely automate the process of creating an amplification honeypot which emulates all discovered vulnerabilities to a degree that is as (if not even more) realistic than honeypot implementations by humans.

Summarizing, we provide the following contributions:

- We present AMPFUZZ, the first systematic approach to discover amplification vulnerabilities in network services based on directed greybox fuzzing augmented with a novel concept of UDP-awareness.

- We evaluate our open-source implementation of AmpFuzz on 25 services from the Debian repositories which we find by searching for network daemons using the SELinux reference policy. AmpFuzz identifies 13 implementations as vulnerable and revealed 4 previously undetected amplification vectors.
- We introduce the first methodology to synthesize honeypot code from the discovered amplification vulnerabilities leveraging a modified symbolic execution engine.

7.4 Background

We first briefly introduce the general concept of fuzzing and briefly review amplification vulnerabilities.

7.4.1 Fuzzing

Fuzzing has become a popular technique to find software vulnerabilities. Originating from the area of software testing, the key idea is to run a System under Test (SuT) under a vast number of random inputs while monitoring it for abnormal behavior. While commonly used to check for crashes, fuzzing has also been successfully applied to finding performance issues [109, 83] or unexpectedly large memory allocations [103].

Blackbox fuzzers generate inputs purely at random and oblivious to the SuT's inner workings. This allows them to test a vast number of inputs quickly, but generally fails on programs that require structured input. Contrarily, whitebox fuzzers [22, 62] try to gain insights into the SuT via static program analysis. For instance, this may entail executing the program symbolically to find constraints on the input for the currently taken path. By partially inverting the constraints, new input that explore previously untaken paths can then be generated by means of a constraint solver. This enables whitebox fuzzing to also reach "deep" parts of the code that cannot be explored through blackbox fuzzing. However, the heavy runtime overhead for symbolic execution, combined with the path explosion problem and insufficient library support renders whitebox fuzzing impractical for many use cases.

Consequently, the most adopted approach to date is (guided) greybox fuzzing [18, 25, 51, 95] aiming at the sweet spot between the aforementioned strategies. Greybox fuzzing augments the scalability of blackbox fuzzing with runtime feedback obtained through lightweight program instrumentation. This runtime feedback is then used to guide the input generation, either towards increasing general code coverage [168, 25, 114] or reaching specific points in the program [24, 27, 52, 57, 133, 160, 102].

7.4.2 Amplification Vulnerabilities

Several UDP-based protocols have known amplification vulnerabilities, including widely-used protocols such as DNS, NTP, or SNMP [120]. Once discovered, these vulnerabilities can be mitigated. For instance, a fatal amplification vulnerability in the Network Time Protocol (NTP) [96] amplified request traffic by a factor of over 550× by abusing an obsolete debugging feature. After its discovery, Kührer et al. coordinated a disclosure of the vulnerability, having reduced the number of amplification by 92% within just 10 weeks [80]. Similarly, the amplification potential in DNS was significantly mitigated by widescale deployment of rate limiting and message truncation [158], or by disabling debugging features [30, 1].

Unfortunately, to date, we lack a systematic way of finding amplification vectors at scale. In fact, amplification vulnerabilities do not resemble typical software bugs. Unlike other security vulnerabilities like memory corruption errors, they do not cause crashes or lead to anomalous software behavior. Instead, most amplification vectors represent (originally) *intended* protocol and/or implementation features, and developers thus have lower awareness of their potential harm. Unfortunately, this also implies that existing fuzzing techniques cannot be simply used to discover amplification vulnerabilities. They resemble an entirely new bug class that fuzzers have never been applied to, and come with unique challenges.

7.5 Fuzzing for Amplification

In this chapter, we aim to find amplification vulnerabilities automatically through fuzzing without relying on a priori knowledge of the underlying network protocols or their implementations. To reach this goal, in principle, we need to find reasonably small UDP requests that trigger larger (amplifying) responses from a given network daemon.

7.5.1 Amplification Fuzzing Challenges

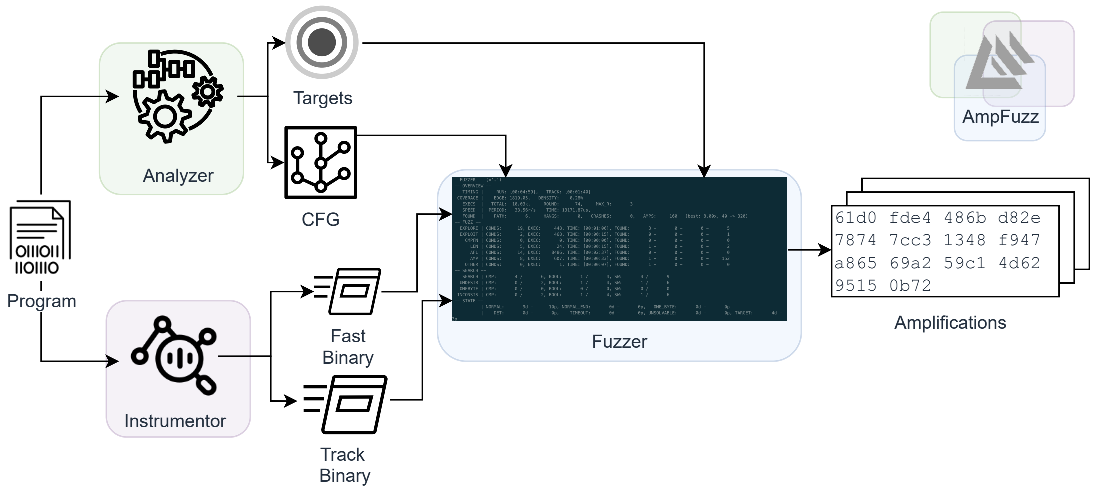
Several challenges hinder the plug-and-play adoption of existing fuzzers to discover amplification vulnerabilities. We will outline these challenges in the following, and come back to how we tackle them in the subsequent sections.

7.5.1.1 Lack of Protocol Knowledge

Network protocol fuzzing is challenging per se, as each protocol comes with its own formats, syntax, and features. When seeking for an automated solution to revealing amplification vectors, we do not want to assume any a priori knowledge of the fuzzing targets. This includes that we do neither rely on protocol specifications, nor want to restrict ourselves to certain implementations of these protocols. While this greatly boosts the generality of our approach, it also implies that we have to rely on some sort of guidance to infer the protocol details (and those of its implementations).

7.5.1.2 Unexplored Vulnerability Class

Amplification vulnerabilities share little similarities to other, well-explored bug classes. A program crash clearly indicates a classical software bug as a binary indicator. Whether or not a certain service is vulnerable to amplification has to be determined through other means. One of the detailed challenges here lies in the fact that a DDoS attacker would typically aim to minimize the request size, while trying to maximize the response size. The success in discovering an amplification vector is thus not a binary decision. Even if amplification can be discovered, there may be ways to further reduce the input, or to change the input to further increase the response size.



```
  FUZZER    (='.')
-- OVERVIEW --
   TIMING |     RUN: [00:04:59],   TRACK: [00:01:40]
 COVERAGE |    EDGE: 1819.05,   DENSITY:    0.28%
   EXECS  |   TOTAL:  10.03k,     ROUND:      74,     MAX_R:       3
   SPEED  |  PERIOD:   33.56r/s    TIME: 13171.87us,
   FOUND  |    PATH:       6,     HANGS:       0,   CRASHES:       0,   AMPS:     160   (best: 8.00x, 40 -> 320)
-- FUZZ --
  EXPLORE | CONDS:      19, EXEC:     448, TIME: [00:01:06], FOUND:       3 -       0 -       0 -       5
  EXPLOIT | CONDS:       2, EXEC:     468, TIME: [00:00:15], FOUND:       0 -       0 -       0 -       1
    CMPFN | CONDS:       0, EXEC:       0, TIME: [00:00:00], FOUND:       0 -       0 -       0 -       0
      LEN | CONDS:       5, EXEC:      24, TIME: [00:00:15], FOUND:       1 -       0 -       0 -       2
      AFL | CONDS:      14, EXEC:    8486, TIME: [00:02:37], FOUND:       0 -       0 -       0 -       0
      AMP | CONDS:       8, EXEC:     607, TIME: [00:00:33], FOUND:       1 -       0 -       0 -     152
    OTHER | CONDS:       0, EXEC:       1, TIME: [00:00:07], FOUND:       1 -       0 -       0 -       0
-- SEARCH --
   SEARCH | CMP:       4 /       6, BOOL:       1 /       4, SW:       4 /       9
  UNDESIR | CMP:       0 /       2, BOOL:       1 /       4, SW:       1 /       6
  ONEBYTE | CMP:       0 /       0, BOOL:       0 /       0, SW:       0 /       0
 INCONSIS | CMP:       0 /       2, BOOL:       1 /       4, SW:       1 /       6
-- STATE --
          | NORMAL:       9d -      10p, NORMAL_END:       0d -       0p,   ONE_BYTE:       0d -       0p
          |    DET:       0d -       0p,    TIMEOUT:       0d -       0p, UNSOLVABLE:       0d -       0p, TARGET:      4d -
3p
```

```
61d0 fde4 486b d82e
7874 7cc3 1348 f947
a865 69a2 59c1 4d62
9515 0b72
```

Figure 7.1: AmpFuzz outline

7.5.1.3 Lack of UDP State

UDP network daemons are a challenging target for fuzzing, since it is non-obvious when the daemon under test has finished processing a request. In the worst case, the request is silently discarded and no response is sent to the fuzzer. And even if the fuzzer receives a response packet, it is unclear whether further response packets will follow. A similar situation occurs during startup of the daemon: At which point is it ready to accept requests from the fuzzer? A request that is sent too early results in an ICMP unreachable packet at best, but could also simply be dropped. This problem is exacerbated by the fact that UDP is a connectionless protocol, such that even on the network layer there is no notion of a failed or terminated connection. While both daemon's startup and replay delays can be addressed by waiting for a fixed amount of time—the de facto workaround used in the literature, e.g., [49]—this slows down fuzzing quite dramatically, as these timeouts have to be awaited before and after each fuzz input.

7.5.2 Design Overview

To tackle the aforementioned challenges, we developed the amplification fuzzing framework AmpFuzz. Figure 7.1 shows our general design. We adopt the overall fuzzing pipeline of ParmeSan [102], a *directed fuzzing* extension of the mutation-based greybox fuzzer Angora [25], which consists of three main components: a static analyzer, a program instrumentor, and the actual fuzzer.

The static analyzer is used to extract a control-flow graph (CFG) as well as a list of targets in the program that the fuzzer should attempt to reach. The program instrumentor instruments the service under test twice, once with only a lightweight coverage collection, and once with a dataflow analysis framework. The former version allows the fuzzer to quickly test whether a new input reaches new and undiscovered parts of the program. The considerably slower dataflow-instrumented version can then be run only on those inputs to provide additional information about which input bytes are relevant for branching decisions. The fuzzer finally

Class	Functions
Source	`recv`, `recvfrom`, `recvmsg`, `recvmmsg`
Sink	`send`, `sendfrom`, `sendmsg`, `sendmmsg`
Blocking	`select`, `pselect`, `poll`, `ppoll`, `epoll_wait`, `epoll_pwait`

Table 7.1: Network function classification

uses the extracted CFG and the list of target locations to prioritize fuzzing inputs and the feedback from the dataflow-instrumented binary to inform its mutation operators.

For AmpFuzz, we modify and extend all three parts of this pipeline to address the challenges mentioned above, as we explain in the following.

7.5.3 Protocol-Agnostic Fuzzing

To fuzz a service for amplification vulnerabilities *without* having a specification of its protocol, we make two observations: First, the service implementation *implicitly* defines the protocol. After a request is received from the network, there will be likely a series of branching conditions that decide whether and how the request should be handled. Second, we only need to follow the protocol close enough to trigger a response from the service. After all, the attacker's goal is to provoke large responses from short requests, regardless of their adherence to a formal protocol specification.

To this end, we build on the directed fuzzing capabilities of ParmeSan [102] and the dataflow analysis of Angora [25]. We define all network functions that receive packets as *sources* and those that send out packets as *sinks* (see Table 7.1). In the static analyzer we collect calls to sink functions as target locations to direct the fuzzer to. During fuzzing, *source* functions serve as starting points for dataflow analysis. Here, Angora's byte-level taint tracking allows the fuzzer to perform targeted mutations of the inputs, eventually producing inputs that are "valid enough" to generate responses.

7.5.4 Amplification Feedback

Yet, program inputs that excite a response are only a first step towards finding amplification vectors. That is, to spot true amplification vectors, we also require a notion of *amplification* in order to find minimal inputs that generate maximal responses. To this end, we build upon the *bandwidth amplification factor* (BAF)—simply put, the response-request ratio in terms of UDP payload sizes—as defined by Rossow [120]:

$$BAF = \frac{\text{len}(\textit{UDP payload amplifier to victim})}{\text{len}(\textit{UDP payload attacker to amplifier})}$$

However, in order to correctly handle services that reply to zero-length packets or that send multiple packets, we chose to also include the upper protocol headers up to the Ethernet layer in the computation. That is, we assume an extra 8 bytes for the UDP header, 20 bytes for the IP header, and 18 bytes for the Ethernet header and trailer, as well as a minimum payload size of 46 bytes for the Ethernet frame:

$$\mathrm{len}_{L2}(x) = 18 + \max(46, 20 + 8 + \mathrm{len}(x))$$

$$BAF_{L2} = \frac{\sum \mathrm{len}_{L2}(\mathit{UDP\ payload\ fuzz\ output})}{\mathrm{len}_{L2}(\mathit{UDP\ payload\ fuzz\ input})}$$

Here, the sum in the numerator is over all UDP packets received from the fuzzed service in reply to a single fuzz input.

In ParmeSan, fuzz inputs were only recorded as new seeds for further mutation if they provided new coverage, i.e., new paths. We extend this and also record inputs that improve the amplification factor of their path. This includes inputs with a $BAF_{L2} \leq 1$ (i.e., no amplification), since subsequent optimization might find an input with a $BAF_{L2} > 1$ for the same path. By keeping a separate maximum amplification ratio *per path*, we can optimize the achieved amplification for each vector independently. For example, in the case of NTP, we would like to find optimal requests for all protocol features such as `monlist` and `read variables`.

We use two techniques to optimize candidate amplification inputs. First, we leverage Angora's dataflow analysis to prioritize input bytes for fuzzing that influence the length of the response packets. Concretely, if the `length` argument of a sending function depends on some of the input bytes, new inputs are generated that mutate those bytes. Second, we add a simple yet effective new mutation operator: Once a valid request has been found through fuzzing, we generate further request candidates by stripping off bytes from the end of the request one by one. This is helpful as many network protocols (or their implementations) ignore trailing bytes, thus the fuzzer has no other incentive to keep its requests short. Furthermore, some also pad incoming requests with NUL-bytes, which consequently can also be omitted.

7.5.5 UDP-Aware Amplification Fuzzing

The lack of UDP state undermines our goal to infer whether or not amplification can occur. Without a clear sign that the SuT finished processing a request, lots of time will be wasted waiting for replies that never happen. On the other hand, amplification-provoking inputs can be falsely ignored if they are sent to the SuT too early and thus dropped on the network layer. We therefore instrument the SuT in order to automatically infer basic UDP network state. Essentially, we want to make the start and the end of request handling observable to the fuzzer. To this end, we define three classes of network functions as shown in Table 7.1: We define all functions that receive a packet as *sources*, all functions that send out a packet as *sinks*, and all functions that can block execution while waiting for a packet as *blocking* functions.

7.5.5.1 Beginning of Request Processing

We chose a shared memory semaphore to notify the fuzzer that the fuzzee is ready to accept requests. For this we hook all calls to functions in the *source* and *blocking* category and unlock the semaphore if the socket to be read or to be waited on is the UDP socket we are

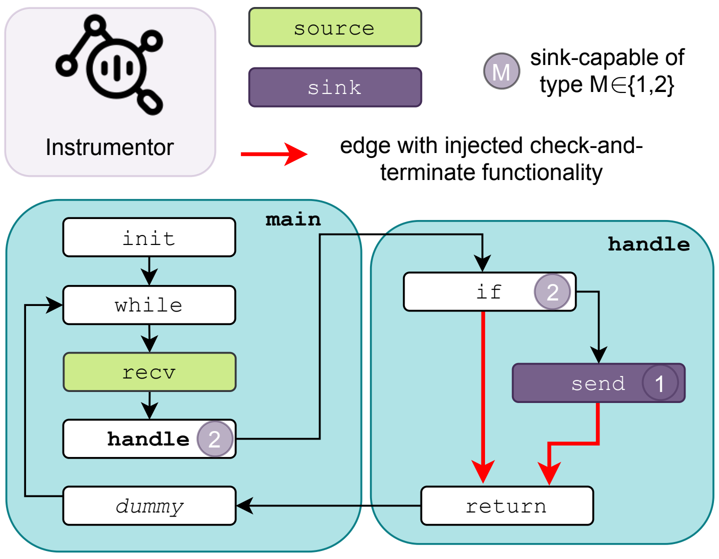

Instrumentor
source
sink
M
sink-capable of type M∈{1,2}
edge with injected check-and-terminate functionality
main
init
while
recv
handle 2
dummy
handle
if 2
send 1
return

Figure 7.2: High-level example of our static analysis execution

currently fuzzing. As a socket's listening state and bound port can be obtained at runtime, no additional socket accounting mechanism is required.

7.5.5.2 End of Request Processing

To signal that request handling has finished, we terminate (parts of) the SuT. We perform this *early termination* in two places. First, we again hook all calls to *source* and *blocking* functions. If the call pertains to our fuzzing socket and would block, we check whether a previous call to a *source* function for that socket had been successful. In that case, we terminate the current thread. Terminating only the current thread but not the entire fuzzee ensures that request handling in other threads can still proceed.

Secondly, we statically analyze the SuT to identify branches that make it impossible to send a response without reading another request first. We will describe our mechanism using Figure 7.2 as a running example. The depicted CFG contains two functions resembling a structure commonly observed in network daemons. That is, function `main` first performs some initialization (`init`), e.g., obtaining a socket, and then starts a loop (`while`) in which it waits for incoming packets (call to *source* `recv`). Once a packet is received, it calls a handling function `handle`, which checks the request (`if`) and either proceeds with a reply (call to *sink* `send`) or returns without (`return`). After the packet has been handled, the loop in `main` starts over waiting for the next packet.

When fuzzing, we would like to terminate the SuT as soon as the current fuzz input has been handled and no further output can be expected. Therefore, we search for paths that lead to areas from which no further *sink* function can be reached without traversing over a *source* function first. Our analysis is based on an inter-procedural CFG built over *strict* basic blocks—function calls also terminate basic blocks and instructions past the call form their own basic block. For simplicity, we also add empty dummy nodes if no instructions immediately follow a call (`dummy` node in `main`). Based on this CFG, we now find edges that can never lead to *sink* functions without passing *source* functions before (→).

For example, consider the `if` basic block in `handle` that calls `send`. When `send` returns, all subsequent paths to this *sink* function will pass `recv`, a *source* function. At these edges, we insert a call to our check-and-terminate code that terminates the current thread if a request had been accepted by the program before.

We can compute the set of these "check-and-terminate" edges using a fixed-point iteration pass over the basic blocks to establish which basic blocks can reach a *sink* function before a *source* function. We will call these *sink-capable*. These are basic blocks that

1. call a *sink* function (e.g., `send` node in `handle`),
2. do not call a *source* function and have at least one successor that is *sink-capable* (e.g., `if` node in `handle` and node with a call to `handle` in `main`).

Undecided basic blocks after this fixed-point iteration are *non-sink-capable*, as they can only remain undecided as part of a loop without outgoing edges to *sink-capable* basic blocks. To ensure that we do not terminate the fuzzee prematurely in presence of dynamic calls or calls to shared libraries, we over-approximate function calls that cannot be analyzed statically as *sink-capable*.

Finally, we add our check-and-terminate functionality to all edges leading from a *sink-capable* basic block to a *non-sink-capable* basic block (denoted as →). In our running example presented in Figure 7.2, these are both incoming edges to the `return` block in function `handle`. Terminating the program at paths that take such edges prevents a fuzzee from persisting in the states which produce no new replies, and consequently, no new amplifications.

7.5.6 Implementation

We implement AMPFUZZ on top of ParmeSan [102] and Angora [25]. As both the analyzer and instrumentor components are implemented as LLVM passes, we use wllvm [162] to compile whole programs to single bitcode files. To enable UDP-aware fuzzing, we extend ParmeSan's instrumentor with our static analysis discussed in Section 7.5.5.

Furthermore, as AMPFUZZ aims at amplification discovery in widespread daemon services, we add support for three important features not handled by ParmeSan. Firstly, we extend the original dataflow tracking mechanism to shared libraries and dynamically loaded code (e.g., plugin systems) used by the daemons extensively. Specifically, we provide a unique branch-ID seed for every object file during instrumentation and hook `dlsym` and `dlopen` functionalities to dynamically load CFGs for libraries. As a result, AMPFUZZ can track input-dependencies even for branch checks inside libraries, ultimately discovering amplifications at speed in cases where the original approach had to resort to randomized input generation. Secondly, we add handling of `fork` by always following the child. This works for several cases in our evaluation, supporting our assumption that `fork` is often used to spawn request handlers. Lastly, we implement a wrapper library for `inetd` services such as `in.tftpd`, as these expect to find the UDP socket as their `stdin` and `stdout`.

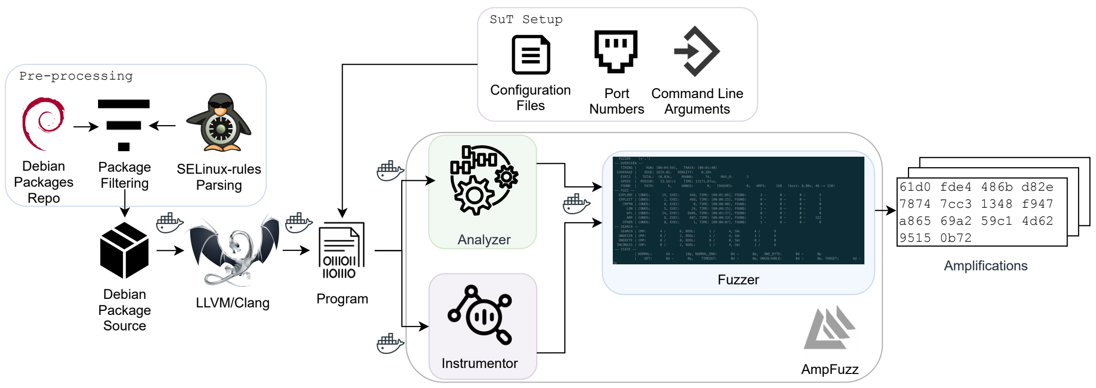


Figure 7.3: Experimental evaluation workflow

7.6 Evaluation

We evaluated AMPFUZZ on 25 programs using the pipeline depicted in Figure 7.3. In a pre-processing step, we collected candidate targets by combining data from the SELinux project [131] and the Debian package repositories [36]. Candidates were then prepared for fuzzing by first recompiling them with wllvm [162] and then performing the fuzzing-specific instrumentation with AMPFUZZ as described in Section 7.5.5. Finally, we provided configuration files and command line arguments for each target program. Due to the manual effort required for this last step, we limited our final selection to 25 programs, but ensured to include packages both with and without known amplification vulnerabilities. To separate fuzz targets from one another and from the host system, we built target-specific docker images containing our fuzzer AMPFUZZ and the instrumented and configured version of the fuzz target.

7.6.0.1 Fuzz Target Selection

To obtain a list of UDP service implementations, we leveraged the SELinux reference policy[1]. In particular, we scanned the policy for file paths which were granted permission to open and handle UDP ports (SELinux labels `corenet_udp_*`). To then find Debian packages that include these files, we used the `apt-file` utility.

As our fuzzing pipeline relies on LLVM-based instrumentation, we next attempted to recompile each package with Clang. We filtered out services which are not written in C/C++ or rely on gcc-specific extensions such as inline assembly.

7.6.0.2 Fuzz Target Configuration

Many network daemons require some form of configuration or setup before they can be executed. This can include specifying command line arguments, editing configuration files, or even run preliminary configuration programs. Even though Debian packages usually attempt to configure a daemon after installation, this was still a manual effort. In order to

[1]https://github.com/SELinuxProject/refpolicy/wiki

prepare a service for fuzzing we thus ensured that the service

1. could be run inside an unprivileged docker container,
2. ran in the foreground whenever possible, and
3. could be started repeatedly in fast succession.

The first point is a consequence of our evaluation setup and could be potentially avoided in the future by fuzzing on the host system directly or using another virtualization mechanism, since it meant that a number of services had to be excluded as they required access to "privileged" functions such as `ioctl`. Ensuring that the service is running in the foreground helps the fuzzer, as it then does not need to track the SuT through multiple `forks`. Finally, since during a fuzzing campaign the service will be restarted multiple times, we ensured that this does not cause unwanted side-effects. In particular, we made sure that the service would not attempt to open connections to the outside world, as these systems would otherwise receive a large number of new connections from our fuzzing system. For example, in the case of the NTP daemon `ntpd`, this meant removing NTP pool entries from the configuration file. If a service could be configured to listen on multiple individual ports, we configured the service for each port individually in a separate docker image.

7.6.0.3 Fuzz Target Instrumentation

Before we proceed with fuzzing, we instrumented each program as detailed in Section 7.5.5. Our instrumentation takes negligible time with respect to the time this additional step saves during fuzzing. Instrumentation takes 6 minutes on average and less than one minute median time per binary. Note that instrumentation makes our approach UDP-aware, which is crucial for the fuzzing performance as we will demonstrate in Section 7.6.3.

7.6.1 Results

We fuzzed each (program, port)-pair for 12 hours on a single CPU core, and repeated the experiment a total of 3 times. Every run was given only a single seed consisting of a single byte `"a"`. Timeouts for individual inputs were set to 5 seconds for normal and 60 seconds for taint-tracking runs. These timeouts were specified to account for the corner case of SuT getting stuck processing the input which was rarely the case thanks to UDP-awareness as detailed in Section 7.6.3. All experiments were performed on a server with 2 Intel® Xeon® Gold 6230N processors and a total of 512 GB of RAM running Debian buster. The target-specific docker images were built on top of the `ubuntu:20.10` base image[2].

The results are shown in Table 7.2. The first three columns specify which program from which Debian package was fuzzed on which port. Column *inputs* denotes the total number of inputs that were generated and tested during the 12-hour fuzzing run, the column *paths* shows how many *new* paths were discovered during fuzzing. For targets where a reflection or amplification vector was found, we also present the mean time until the first reply-inducing candidate was found (*μt 1st*), the mean time until the maximum amplification was found (*μt max.*), and the maximum Ethernet bandwidth amplification factor (*BAF_{L2}*) of the best

[2]`sha256:671495eee4d808a4c78517c3be67355ae0afef2796157cbf42e1efcd0f2a1ccf`

	package	program	port	inputs	new paths	μt 1st	μt max.	BAF_{L2}	novel		
known vulnerable	atftpd	atftpd	69	119 661	1	0s	0s	**1.14**		✓	✓
		in.tftpd	69	104 397	1	0s	0s	**1.14**		✓	✓
	knot	knotd	53	6 972	14	17m52s	3h50m	1.00			
	minissdpd	minissdpd	1900	118 105	108						
	ntp	ntpd	123	66 866	39	15m32s	1h19m	**7.03**		✗	
	ntpsec	ntpd	123	63 017	23	41m30s	41m30s	1.00			
	openbsd-inetd	inetd	19	116 727	1						
	quagga-ripd	ripd	520	7 048	7						
	tftpd	in.tftpd	69	120 138	1	2m24s	2m24s	**1.14**		✓	✓
	xinetd	xinetd	19	74	0	2s	2s	**16.72**		✓	✓
unknown	inetutils-syslogd	syslogd	514	100 995	42						
	krb5-admin-server	kadmind	464	20 054	284						
	krb5-kdc	krb5kdc	88	33 689	248						
	openafs-fileserver	bosserver	7007	62 097	7	32s	36m15s	**4.59**	✓	✓	✓
	openbsd-inetd	inetd	7	121 655	1						
			13	116 781	1						
			37	121 705	1						
	rsyslog	rsyslogd	514	5 959	104						
	talkd	in.ntalkd	518	132 907	0	0s	0s	**1.09**	✓	✓	✓
		in.talkd	517	129 487	0	0s	0s	**1.09**	✓	✓	✓
	xinetd	xinetd	7	68	0	2s	2s	1.00			
			9	73	1						
			13	74	0	2s	2s	**1.12**	✓	✓	✓
			37	74	0	2s	2s	1.00			
	xl2tpd	xl2tpd	1701	133 918	17						

Table 7.2: Results for Debian package fuzzing, honeypot synthesis and its testing, grouped by services with known (top) and unknown (bottom) amplification vectors.

run. Lastly, we split the table between services with known amplification vulnerabilities and those without. For the latter, *novel* indicates new and previously unknown vulnerabilities.

Overall, we find that AmpFuzz discovers amplification or reflection vulnerabilities in 13 of the 25 tested daemons, 9 of which are *true* amplifications ($BAF_{L2} > 1$). More interestingly, AmpFuzz identifies 6 previously unknown reflection and amplification vulnerabilities. Next to trivial reflections in the legacy echo (7), daytime (13), time (37) and talk (517)/ntalk (518) protocols, these also include a non-trivial amplification in the Basic OverSeer Server of OpenAFS, which we will detail in the next section.

Our evaluation also includes 10 services for which amplification vulnerabilities had been found before through manual analysis. On these, AmpFuzz manages to find 5 true amplifications and 2 reflections. We investigated the three cases in which AmpFuzz fails to find an expected amplification: For minissdpd, we find that the service in the given configuration does indeed not respond to `M-SEARCH` requests, and is therefore invulnerable to the known amplification vector. In the case of quagga-ripd and openbsd-inetd, our evaluation setup is at fault: These services ignore all requests from the loopback interface, which we use for communication between the fuzzer and the SuT. For openbsd-inetd, we verified that AmpFuzz succeeds to find the expected amplifications after manually disabling the check.

AmpFuzz is also reasonably fast in vulnerability discovery. All vulnerabilities could be found in less than four hours, and the vast majority within a few minutes. Furthermore, after having discovered all relevant paths (if any), AmpFuzz gracefully terminated itself. For instance, fuzzing `xinetd` terminated after having explored all network-related paths with ≤ 74 inputs.

7.6.2 Case Study: OpenAFS `bosserver`

AmpFuzz identifies a new amplification vector in the Basic OverSeer (BOS) Server of the OpenAFS distributed filesystem. This server is responsible for monitoring other processes of the AFS filesystem and offers a UDP interface on port 7007. The protocol employed by the BOS server uses packets with a fixed-size 28-byte header followed by a variable-length payload (shown in Listing 7.1).

Packets of type `RX_PACKET_TYPE_DEBUG` and with the `RX_CLIENT_INITIATED` flag set can be used to query for debug packets. Setting payload type `RX_DEBUGI_RXSTATS` further specifies a communication statistics query, which produces a 312 bytes response.

AmpFuzz can find all of these constraints through its dataflow-assisted fuzzing. Furthermore, our added mutation operator, which shortens the request, allows AmpFuzz to generate requests that omit the last four bytes (`rx_debugIn.index`), as they are irrelevant in that case. The shortest request payload is thus only $28 + 4 = 32$ bytes in size. This results in a UDP payload BAF of 9.75 (BAF_{L2} 4.59), which is higher than the amplification potential of other, widely-abused protocols such as SNMP or NetBios [120].

```
/ HEADER /
struct rx_header {
    afs_uint32 epoch;       / Start time of client process /
    afs_uint32 cid;         / Connection id (defined by client) /
    afs_uint32 callNumber;  / Current call number /
    afs_uint32 seq;         / Sequence number within this call /
    afs_uint32 serial;      / Serial number of this packet /
    u_char type;            / RX packet type /
    u_char flags;           / Flags, defined below /
    u_char userStatus;      / User defined status information /
    u_char securityIndex;   / Service—defined security method to use /
    u_short serviceId;      / Service this packet is directed _to_ /
    u_short spare;          / Now used for packet header checkksum. /
};

/ DEBUG PAYLOAD /
struct rx_debugIn {
  afs_int32 type;           / requested type; range 1..5 /
  afs_int32 index;          / record number: 0 .. n /
};
```

Listing 7.1: Packet structure used by OpenAFS bosserver

7.6.2.1 Coordinated Disclosure

We contacted the maintainers of OpenAFS about our newly found amplification vector on 4th of June 2021 and asked for their permission to publish our findings after a three-month period, following the industry best practices. They promptly confirmed our findings and approved our proposed timeframe for publication. Interestingly, they informed us that this particular amplification vulnerability found by AMPFUZZ not only affects the BOS Server, but all OpenAFS services which share the same underlying RX RPC mechanism (which were, however, not part of our evaluation).

To estimate the number of vulnerable services, we performed an Internet-wide scan for UDP port 7007, which is used by the BOS Server. Our scan revealed just shy of 1k vulnerable OpenAFS BOS Server instances. However, the OpenAFS maintainers had mentioned that BOS Server instances are usually run behind a firewall, since they require no external communication. Unfortunately, such firewalling is not possible for a number of other OpenAFS services, which are similarly affected. Indeed, further scans including port numbers of other affected OpenAFS services (7000-7003, 7005) indicate that there is a total of around 16k vulnerable OpenAFS devices in IPv4, more than enough to launch serious attacks.

7.6.3 Impact of UDP-Aware Fuzzing

Lastly, we also evaluate the impact of UDP awareness on fuzzing progress. As mentioned in Section 7.5.5, without UDP awareness, the fuzzer cannot know when request processing has finished and thus has to resort to waiting for the timeout to expire. To this end, we ran AMPFUZZ twice on the 25 configured services, once with UDP awareness and once without. We let each run fuzz for ten minutes, using the same timeouts as before (5s/60s).

binary (pkg)	port	execs/sec		**speed up**
		unaware	aware	
atftpd	69	0.06	38.24	**609**×
in.tftpd (atftp)	69	0.07	29.34	**449**×
syslogd	514	0.02	104.54	**6 934**×
knotd	53	0.05	0.04	**1**×
kadmind	464	0.08	1.33	**17**×
krb5kdc	88	0.08	0.38	**5**×
minissdpd	1900	0.03	113.17	**4 095**×
ntpd (ntp)	123	0.02	79.26	**3 380**×
ntpd (ntpsec)	123	0.05	78.58	**1 443**×
bosserver	7007	0.06	41.85	**724**×
inetd	7	0.17	100.87	**590**×
	13	0.17	100.91	**591**×
	19	0.17	100.73	**590**×
	37	0.17	100.83	**590**×
ripd	520	0.14	0.17	**1**×
rsyslogd	514	0.01	0.01	**1**×
in.ntalkd	518	0.13	106.75	**817**×
in.talkd	517	0.14	106.78	**741**×
in.tftpd (tftpd)	69	107.32	107.07	**1**×
xinetd	7	0.14	4.26	**31**×
	9	0.14	4.26	**31**×
	13	0.10	4.25	**43**×
	19	0.14	4.20	**31**×
	37	0.14	4.24	**31**×
xl2tpd	1701	0.08	118.80	**1 525**×

Table 7.3: UDP-Aware vs UDP-Unaware Fuzzing

Table 7.3 shows the average fuzzing throughput in executions per second for the UDP-*unaware* and the UDP-*aware* versions of AmpFuzz. As shown in the last column, a fuzzer that does not need to wait for timeouts can test new inputs more efficiently, in many cases achieving a hundred- or thousandfold speedup over its UDP-unaware counterpart. This clearly highlights the needs and benefits of UDP-aware fuzzing.

7.7 Honeypot Synthesis

On the defensive side, next to coordinated disclosure efforts, amplification honeypots have proven as an invaluable tool. By mimicking the behavior of vulnerable systems, they strive to be discovered by attackers and included as reflectors in subsequent attacks. As such, they not only allow to monitor and study attacks in real-time [S1, 153], but also form the basis of our and others' traceback mechanisms [P1, P2, P3, 50].

However, creating such honeypot systems demands substantial manual effort. Since running

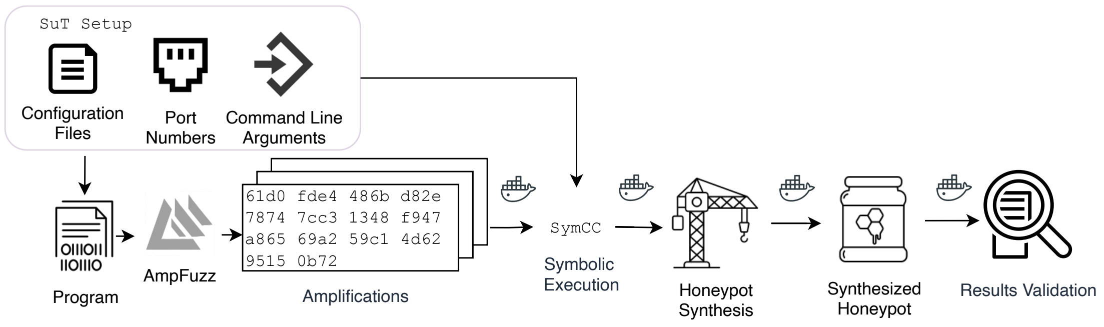


Figure 7.4: Honeypot code synthesis outline

full implementations of the vulnerable services would introduce prohibitive overhead, lightweight replica implementations of their request-response behavior are required. For every amplification vector analysts thus need to determine to which requests the honeypot should respond and how the response should be computed.

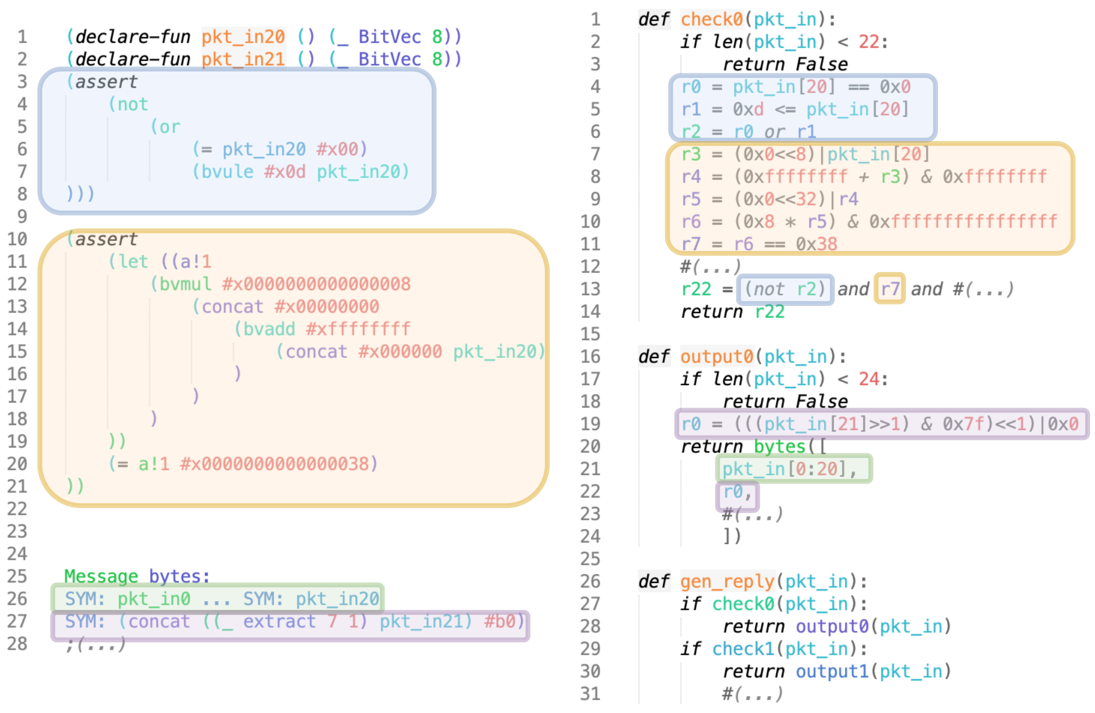
In this section we therefore present our approach to automated honeypot synthesis based on AMPFUZZ.

7.7.1 Honeypot Synthesis Overview

Figure 7.4 outlines our honeypot synthesis workflow. In essence, we use symbolic execution to capture path constraints on the request and a symbolic expression of the generated response. From this we then derive implementations of a `check`-function, which matches the constraints against the current request, and an `output`-function, which produces a corresponding response. The synthesized honeypot thus consists of a collection of (`check`, `output`)-pairs. On a new input, this collection is traversed and the `check` functions are executed one by one. If a `check` function shows a match for the current input, its `output` is then used to generate the response.

7.7.1.1 Symbolic Execution

To collect path constraints and output expressions, we leverage the state-of-the-art LLVM symbolic execution framework SymCC by Poeplau and Francillon [111]. SymCC can be nicely integrated with AMPFUZZ, since both rely on LLVM IR for instrumentation. We extend SymCC by providing symbolic wrappers for receiving and sending network functions. For receiving functions, we generate new symbolic bytes for every byte read from correct UDP socket. This allows SymCC to treat network requests as symbolic inputs. For sending functions, we dump all collected path constraints, but also output the symbolic expression for every message byte. At this point, the path constraints capture exactly which conditions the request has to fulfill in order for the current response to be sent, while the output expressions capture how the individual bytes of the response are computed. Hence, to generate constraints and expressions for a specific amplification vector, we only need to replay the amplification request found by AMPFUZZ against the SymCC-instrumented version of the target service.



```
1   (declare-fun pkt_in20 () (_ BitVec 8))
2   (declare-fun pkt_in21 () (_ BitVec 8))
3   (assert
4       (not
5           (or
6               (= pkt_in20 #x00)
7               (bvule #x0d pkt_in20)
8   )))
9
10  (assert
11      (let ((a!1
12          (bvmul #x0000000000000008
13              (concat #x00000000
14                  (bvadd #xffffffff
15                      (concat #x000000 pkt_in20)
16                  )
17              )
18          )
19      ))
20      (= a!1 #x0000000000000038)
21  ))
22
23
24
25  Message bytes:
26  SYM: pkt_in0 ... SYM: pkt_in20
27  SYM: (concat ((_ extract 7 1) pkt_in21) #b0)
28  ;(...)
```

```
1   def check0(pkt_in):
2       if len(pkt_in) < 22:
3           return False
4       r0 = pkt_in[20] == 0x0
5       r1 = 0xd <= pkt_in[20]
6       r2 = r0 or r1
7       r3 = (0x0<<8)|pkt_in[20]
8       r4 = (0xffffffff + r3) & 0xffffffff
9       r5 = (0x0<<32)|r4
10      r6 = (0x8 * r5) & 0xffffffffffffffff
11      r7 = r6 == 0x38
12      #(...)
13      r22 = (not r2) and r7 and #(...)
14      return r22
15
16  def output0(pkt_in):
17      if len(pkt_in) < 24:
18          return False
19      r0 = (((pkt_in[21]>>1) & 0x7f)<<1)|0x0
20      return bytes([
21          pkt_in[0:20],
22          r0,
23          #(...)
24          ])
25
26  def gen_reply(pkt_in):
27      if check0(pkt_in):
28          return output0(pkt_in)
29      if check1(pkt_in):
30          return output1(pkt_in)
31          #(...)
```

Figure 7.5: Honeypot code example for OpenAFS bosserver

7.7.1.2 Code Synthesis

As a result of the previous step, we obtain an SMT-LIB [12] model with a set of assertions describing how to validate the amplification input and a list of expressions, one for each byte of the corresponding reply. In order to build a lightweight honeypot that does not rely on expensive SMT solvers to evaluate these, we instead generate Python code equivalent to the model. To this end, we convert all expressions of the model into a single-static-assignment form by performing a post-order traversal of the expressions' ASTs. During traversal, operators and constants are replaced by their Python counterparts, while a cache ensures that equivalent subtrees are converted only once. Since SMT-LIB expressions operate on fixed-width data types, we introduce additional Python expressions to truncate values to their expected bit-length.

Figure 7.5 shows an example of the honeypot code generation for the newly found amplification vulnerability in OpenAFS' `bosserver`, with parts of the model on the left and their corresponding honeypot code on the right. Note that we only display a fraction of the original codebases here as they are of significant sizes (28.2k LoC for model and 4.7k LoC for honeypot respectively). Lines 10-21 demonstrate an example of a request filtering constraint, which includes bitvector concatenation and bitvector arithmetic operations; its corresponding honeypot check code is shown in lines 7-11. In case the check is successful, a corresponding output function is called. Replies are synthesized using the message bytes description we got from the symbolic execution. In the example of `bosserver`, line 26 of the model tells us that the output's first 20 bytes should be the same as the first 20 input bytes. However, byte 21 should be modified, as shown in line 27. In particular, the output byte is computed as the 7 highest-order bits of the 21st input byte with an appended zero bit. For our generated honeypot code, this translates to a sequence of shifts and bitwise operations, resulting in the expression given in line 19. Lastly, the output function returns the generated reply as a sequence of bytes, which are then sent as a UDP packet to the originator of the request by our synthesized honeypot.

7.7.2 Synthesized Honeypot Evaluation

As depicted in Table 7.2 we were able to synthesize honeypot code (📄) for all discovered amplification vulnerabilities, with the sole exception of `ntpd`, for which the compiler emitted LLVM IR instructions currently not supported by SymCC. To ensure that our synthesized honeypot works as intended, we compared its responses to those of the original services (🔍). Concretely, we replayed the amplifications systematically discovered by AMPFUZZ once against the original service implementation, and once against our honeypot. In all cases as shown in Table 7.2 did the honeypot generate a response when the original service did. Furthermore, responses by the honeypot were always of equal length to those of the original service. While in many cases responses between the two were even indistinguishable on a byte-level, we also observed variance in others. This is expected for services that include random or varying data in their responses like, e.g., session identifiers, but can also appear as an artifact of concretization. Such concretization can occur whenever non-SymCC-instrumented code such as external libraries affects the current execution path, as we will discuss in Section 7.8.2. Overall, this shows that once amplification vulnerabilities are found, honeypot code can be reliably synthesized. In contrast to human-written honeypots, the emulation is free of errors and more realistic, as it accurately reflects the software's behavior.

7.8 Discussion

In this section, we outline shortcomings of our evaluation and how they can be tackled, describe limitations of our honeypot synthesis based on symbolic execution, and describe how we adhere to best ethical standards during our active measurements and by disclosing the vulnerabilities to vendors.

7.8.1 Evaluation Shortcomings

While the set of services we analyzed with AMPFUZZ was limited mostly due to the need for manual configurations, a few services also had to be excluded for technical reasons.

7.8.1.1 LLVM IR

Both the underlying ParmeSan fuzzer and the newly added extensions for UDP-aware fuzzing rely on LLVM IR for target instrumentation. This means that services can be fuzzed only if they can be compiled using an LLVM-based toolchain. For our implementation we further relied on wllvm [162], thus restricting our dataset to services written in C/C++. Fortunately, this includes most network daemons on Linux. However, we noted a few select cases where services were implemented in scripting languages such as Perl or used gcc-specific extensions such as inline assembly.

7.8.1.2 UDP Sockets

Another limitation stems from our choice of using "real" UDP sockets for passing in- and output between the fuzzer and the SuT. While this ensures that all socket-related APIs, especially those relying on socket state such as `poll` and `select`, behave as they would in the real-world scenarios, measures have to be taken to separate SuTs from the host system and from one another, e.g., to avoid conflicts.

To this end, we used Docker containers to isolate different SuTs into their own namespaces. However, without granting additional privileges to these containers, access to some low-level system calls is restricted. We thus had to exclude a number of targets that, e.g., attempted to perform additional socket configuration using `ioctl` calls. Real sockets also impact parallelization during fuzzing, as only one socket may be bound to the same port and address at a time.

Both problems could potentially be avoided by preventing the SuT from binding "real" sockets and hooking the relevant socket API functions instead, albeit at the cost of a more involved instrumentation. Additionally, fuzzing speed could also be increased by having individual SuT instances bind to different addresses in the 127.0.0.0/8 range, as long as the SuT can be configured accordingly and exclusive access to other resources is not required.

7.8.1.3 Source Addresses

Related to the use of actual UDP sockets, we also noticed that some daemons ignore requests from local addresses, while others might ignore everything else. This could be solved by either manual inspection of the SuT or by extending AMPFUZZ with functionality that tries fuzzing both from local and non-local addresses.

7.8.2 Symbolic Execution

Our honeypot synthesis makes use of symbolic execution in order to collect path constraints on the input and symbolic expressions of the generated output. Since we base our synthesis on SymCC, we also inherit some of its technical limitations. As SymCC also relies on LLVM IR, the restrictions mentioned in Section 7.8.1.1 also apply here.

7.8.2.1 Concretization

Furthermore, the expressions generated by SymCC can be incomplete in some cases. For example, calls to libraries that were not compiled with SymCC inevitably lead to a loss of symbolic information. The same applies to yet unsupported LLVM IR instructions. In these cases, only the *concrete* values during execution can be retained.

For this reason, our synthesized honeypot might miss additional checks on the input and generate responses even in cases where the original vulnerable service would not. Likewise, if parts of the output could only be captured concretely, our honeypot might respond with fixed byte sequences instead of *input-dependent* expressions. These shortcomings are, however, comparatively small when looking at the state-of-the-art with manually-written DDoS honeypots which can easily be recognized as such due to missing features and hard-coded responses.

7.8.2.2 Constraint Solving for Fuzzing

Lastly, leveraging the symbolic constraints found through SymCC already at the fuzzing stage might yield further improvements of AMPFUZZ. Such hybrid fuzzing approaches, introduced by Driller [143], can potentially reach "deeper" code paths into the SuT by using constraint solvers to find new inputs. However, as the added computation time required for constraint solving reduces the total number of execution that can be achieved, hybrid fuzzing can perform worse in some cases, where simpler methods like random fuzzing or taint-tracking suffice.

7.8.3 Active Measurements

To assess the prevalence of vulnerable systems, and hence the threat posed by amplification vulnerabilities discovered with AMPFUZZ, we performed Internet-wide scans. While conducting scans we followed best practices [44] in order to ensure that our experiments caused no harm: We only scanned a significant number of randomly sampled IP addresses to be able to extrapolate meaningful results, sent out only a single packet per destination, and obeyed our institute's established blocklist to exclude networks from the scan that had asked us to. In addition, we also made sure that our probes had no ill effects on the target systems through local experiments and source code reviews. For example, in the case of OpenAFS, we concluded that the debug packets we used had no side effects other than incrementing a statistics counter.

7.8.4 Coordinated Disclosure

Where possible, we contacted the maintainers of affected packages and reported the vulnerabilities. We also asked for permission to disclose our findings after a minimum of 90 days, in line with the industry standard. No party asked us to redact our results.

7.9 Related Work

Amplification DDoS and (network) fuzzing have both been active fields of research in the past. While prior works pertaining to amplification DDoS are discussed in Chapter 3, here we detail how previous works from the area of fuzzing relate to AMPFUZZ. A notable exception is AmpMap by Moon et al. [97], which also utilizes fuzzing techniques to find amplification vulnerabilities and hence deserves an in-depth discussion in the context of AMPFUZZ.

7.9.1 AmpMap

The goal of AmpMap [97] is to provide an estimate of the global "amplification risk", considering that a server's amplification factor depends on the software running on the server as well as its current configuration, and further taking into account that a protocol may suffer from multiple amplification vulnerabilities triggered by different requests. For this, AmpMap obtains a list of servers running a certain protocol from public scan services like Shodan or Censys, and then probes these servers in three stages: First, grammar-based random blackbox fuzzing is used to find any amplification inducing requests, which are then replayed against all servers in the second stage. Finally, individual query fields are varied to find additional amplification query variants.

AMPFUZZ differs from this in two key aspects: Firstly, the goal of AMPFUZZ is not to quantify the global amplification potential, but to support software developers and package maintainers in identifying new amplification vulnerabilities proactively. Secondly, although AmpMap does employ blackbox fuzzing, it requires detailed protocol descriptions beforehand in order to generate requests. This also implies that it cannot identify amplifications resulting from malformed or illegal packets. AMPFUZZ on the other hand employs greybox fuzzing to generate packets that excite a response from the SuT, regardless of their protocol adherence.

7.9.2 Network Fuzzing

With the recent trend of software fuzzing, also a number of fuzzers have been developed which can target network daemons. Next to general-purpose fuzzers that simply use network sockets as other means of providing input to the SuT [61, 101, 3, 69, 37], this includes dedicated network fuzzers which generate inputs either based on previously recorded client-server interactions [121, 148, 55] or from protocol descriptions [139, 49, 107]. Some further attempt to infer server-side state in order to reach code paths that require multiple messages between a client and the server [55, 47, 110]. Most of these only target TCP services, where terminated connections can be observed easily, and the ones which allow for fuzzing UDP services either ignore replies from the server completely [3, 69], or rely on timeouts [107, 101, 61, 121, 148, 110, 49] or user-provided target-specific scripts [37]. However, as shown in Section 7.6.3, UDP-awareness of AMPFUZZ outperformed simple timeout-based solutions by multiple orders of magnitude. More importantly though, the goal of these previous fuzzers is finding either inputs which lead to server-side crashes or to detect differences between a protocol's specification and implementation. AMPFUZZ on the other hand is concerned with finding amplification vulnerabilities in a greybox, yet protocol-agnostic way.

7.10 Conclusion

In this chapter, we presented AMPFUZZ, the first approach to systematically discover amplification DDoS vulnerabilities in UDP-based network services. To this end, AMPFUZZ leverages the advancements in directed greybox fuzzing to discover inputs that trigger large network service responses. Moreover, AMPFUZZ augments fuzzing with UDP-awareness, i.e., the ability to distinguish different protocol states, by combining dynamic instrumentation with a static pre-processing. Our experiments on real-life network services show that UDP-awareness leads up to a thousandfold increase in executions-per-second. We conducted our evaluation on 25 daemons extracted from the Debian package repositories, after finding candidate daemons through an analysis of the SELinux reference policy. In total, AMPFUZZ identified amplification or reflection vulnerabilities in 13 network services with 9 *true* amplifications ($BAF_{L2} > 1$). Next to rediscovering 5 known vulnerabilities, which had been found through painstaking manual analysis, our principled approach revealed 6 previously unidentified vulnerabilities, the most severe of them with a non-trivial 4.59 BAF_{L2}.

Furthermore, this chapter shows how to automatically synthesize amplification honeypots from the discovered vulnerabilities. Specifically, we leverage a state-of-the-art symbolic execution engine to produce a model of the amplification vulnerability, which is then converted into executable Python code. To the best of our knowledge, this is the first automated approach to generalize amplifications to honeypots, streamlining proactive and reactive defenses.

8
Conclusion

	Applicability	A Priori Knowledge	Granularity
Scanner Fingerprinting (Chapter 4)	scan = attack infrastructure	-	origin network (scanner IP)
Booter Identification (Chapter 5)	attacks from known booter services	self-attacks	booter service
BGPEEK-A-BOO (Chapter 6)	long-term attack sources	(AS relationship data)	origin AS

Table 8.1: Comparison of the presented traceback mechanisms

8.1 Summary of Contributions

Amplification DDoS attacks pose a substantial threat to Internet participants, allowing attackers to easily generate large amounts of traffic that can render target systems unaccessible. At the same time, defensive approaches are limited to reactive remediation efforts, as the true perpetrators are hidden behind a veil of IP spoofing. In this dissertation, we presented a line of work that aims to tip the scale back in favor of the defenders. Specifically, we answer our two research questions, namely **(RQ1) How can we provide practical traceback for amplification attacks that does *not* require the cooperation of multiple external parties?** and **(RQ2) How can we find amplification vectors and update defensive tooling *proactively*?**

8.1.1 Practical Traceback (RQ1)

Finding the true network origin of such attacks is a key enabler for prosecuting the perpetrators, a crucial complementary step towards mitigating the threat posed by amplification DDoS. To answer our first research question, we provided three honeypot-based attribution approaches that aim at different tradeoffs between applicability, required a priori knowledge, and the resulting traceback granularity as shown in Table 8.1.

First, in Chapter 4, we introduced a fingerprinting technique that enables us to link attacks to the systems which were used for scanning in preparation for the attack. By responding to incoming requests only selectively, based on the request's source address and the address of the honeypot, we can ensure that every scanner can only find a unique subset of our honeypots. This set then serves as a fingerprint which we can compare against the subset of honeypots abused in a particular attack. Using this technique we were able to attribute over 58% of attacks with a confidence of over 99.9%. Regardless of its network location, the scanner could already be considered an accomplice of the attack. However, using TTL-based trilateration we further found evidence that in many cases the same infrastructure is used for scanning and attacking, which allowed us to identify 34 networks as attack *sources* at 98% certainty.

Our second approach, presented in Chapter 5, showed that it is possible to link multiple attacks launched from the same source after training a classifier. In particular, we considered attacks from booter services. By requesting attacks against ourselves, we were able to learn characteristic attack features, such as the selection of honeypots, the number of unique source ports, and the observed TTL distributions, on which we could train a k-NN classifier. In multiple experiments we then confirmed that our classifier can be tuned to achieve almost no incorrect attribution (precision over 99%), while still sustaining a good attribution rate (recall over 69%).

Third, in Chapter 6, we presented a more general traceback mechanism that can find the attacking network without the need for self-attacks and even if scan and attack infrastructures are disjoint. Here, we exploit the fact that attack packets, even when spoofed, must be routed from their source network to the honeypot. We can leverage BGP Poisoning to make our honeypots temporarily inaccessible by certain networks, and correlating this with the observed attack traffic allows us to infer whether the blocked networks were attack sources. We show that BGP-based traceback is feasible in principle with no further knowledge and can find an attack source in less than four days, but that it can be significantly sped up if AS relationship data is available—to under five hours in many cases.

8.1.2 Proactive Discovery of Amplification Vectors (RQ2)

In addition to these traceback techniques we also consider the underlying problem of finding amplification vulnerabilities and deriving honeypot implementations. Until now, both have largely relied on manual analysis, and oftentimes vulnerabilities became known only after they were abused in attacks. Furthermore, honeypots play a central role in all of our traceback approaches, and adding timely support for new amplification vectors is crucial for accurate attack monitoring. To answer our second research question, we therefore presented the first systematic approach to finding such vulnerabilities through guided fuzzing in Chapter 7. Through a combination of static pre-processing and dynamic instrumentation our fuzzer can observe when the service under test is ready to receive a network packet and when processing a request has finished. This enables us to fuzz UDP services efficiently with a thousandfold speed-up over timeout-based solutions. In an evaluation on 25 services from the Debian repositories our tool rediscovered five known amplification vulnerabilities and found four new ones. Finally, we show how a modified symbolic execution engine can be leveraged to automatically synthesize honeypots for the found vulnerabilities, resulting in a fully automated end-to-end pipeline.

8.2 Future Research Directions

IP spoofed traffic is a concern also outside the context of amplification DDoS attacks, as it enables a wide range of attacks, from DNS cache poisoning [8] to stealth partitioning attacks against cryptocurrencies [154]. Consequently, finding the true source of spoofed traffic is also of interest in these settings. While our traceback approaches presented in Chapter 4 and Chapter 5 inherently rely on features of amplification DDoS attacks, our BGP-based methodology from Chapter 6 does not and could hence be adapted to other scenarios. Furthermore, existing IP traceback mechanisms such as packet marking [127, 42, 126, 140, 35, 164, 45, 43, 132, 165, 26, 54] or flow telemetry [137, 138, 87, 75, 147] and our approaches can be seen as opposite extremes with regards to deployment requirements. While the former require an Internet-wide deployment, our techniques require no external collaboration and can be deployed by a single party. This begs the question whether hybrid approaches are possible, e.g., a honeypot-based system that is supplemented by traffic measurements from a number of cooperating ISPs. For example, if netflow data confirms that spoofed traffic traverses a certain ISP, the search space for BGPEEK-A-BOO can be pruned to the subgraph behind that ISP.

In another line of future research, the systematic search of amplification vulnerabilities that we started with AMPFUZZ (Chapter 7) should be furthered. While fuzzing is well suited to finding amplification vulnerabilities in stateless services, a number of known vectors abuse services that maintain state. For example, the infamous Memcached amplification vector [31] makes use of storing a large blob of data first, which is then later requested for during the attack. Likewise, DNS amplification depends not only on the configuration of the tested resolver, but also on the records stored for the queried domain at its authoritative server. Both of these cannot be found with existing methods. Another challenge for vulnerability discovery is closed source software. This is relevant in particular for amplification attacks, where a number of recent amplification vectors were found in Microsoft services, such as connectionless LDAP [167] and the Microsoft Remote Desktop Protocol [41]. As such, combining automated amplification vulnerability search with symbolic execution and binary instrumentation techniques can help to rid the Internet of vulnerable systems—possibly before they cause new attack bandwidth records.

Bibliography

Author's Papers for this Dissertation

[P1] **Krupp, J.**, Backes, M., and Rossow, C. Identifying the Scan and Attack Infrastructures Behind Amplification DDoS Attacks. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, 1426–1437. URL: https://doi.org/10.1145/2976749.2978293.

[P2] **Krupp, J.**, Karami, M., Rossow, C., McCoy, D., and Backes, M. Linking Amplification DDoS Attacks to Booter Services. In: *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*. Springer. 2017, 427–449. URL: https://doi.org/10.1007/978-3-319-66332-6_19.

[P3] **Krupp, J.** and Rossow, C. BGPeek-a-Boo: Active BGP-based Traceback for Amplification DDoS Attacks. In: *6th IEEE European Symposium on Security and Privacy*. 2021. URL: https://publications.cispa.saarland/3372/.

[P4] **Krupp, J.**, Grishchenko, I., and Rossow, C. AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities (and Using Those to Synthesize DDoS Honeypots). 2021. Under submission (USENIX Security 2022).

Further Contributions of the Author

[S1] Krämer, L., **Krupp, J.**, Makita, D., Nishizoe, T., Koide, T., Yoshioka, K., and Rossow, C. AmpPot: Monitoring and Defending Against Amplification DDoS Attacks. In: *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*. Springer. 2015, 615–636. URL: https://doi.org/10.1007/978-3-319-26362-5_28.

[S2] **Krupp, J.**, Schröder, D., Simkin, M., Fiore, D., Ateniese, G., and Nürnberger, S. Nearly Optimal Verifiable Data Streaming. In: *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*. 2016, 417–445. URL: https://doi.org/10.1007/978-3-662-49384-7_16.

[S3] Fleischhacker, N., **Krupp, J.**, Malavolta, G., Schneider, J., Schröder, D., and Simkin, M. Efficient Unlinkable Sanitizable Signatures from Signatures with Re-randomizable Keys. In: *Public-Key Cryptography - PKC 2016 - 19th IACR International Conference on Practice and Theory in Public-Key Cryptography, Taipei, Taiwan, March 6-9, 2016, Proceedings, Part I*. 2016, 301–330. URL: https://doi.org/10.1007/978-3-662-49384-7_12.

[S4] Döttling, N., Fleischhacker, N., **Krupp, J.**, and Schröder, D. Two-Message, Oblivious Evaluation of Cryptographic Functionalities. In: *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part III*. 2016, 619–648. URL: https://doi.org/10.1007/978-3-662-53015-3_22.

[S5] Jonker, M., King, A., **Krupp**, **J.**, Rossow, C., Sperotto, A., and Dainotti, A. Millions of targets under attack: a macroscopic characterization of the DoS ecosystem. In: *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017*. 2017, 100–113. URL: https://doi.org/10.1145/3131365.3131383.

[S6] **Krupp**, **J.** and Rossow, C. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In: *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 2018, 1317–1333. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp.

[S7] Toorn, O. van der, **Krupp**, **J.**, Jonker, M., Rijswijk-Deij, R. van, Rossow, C., and Sperotto, A. ANYway: Measuring the Amplification DDoS Potential of Domains. In: *17th International Conference on Network and Service Management, CNSM 2021, Izmir, Turkey, October 25-29, 2021*. 2021.

Other References

[1] Abley, J., Guðmundsson, Ó., Majkowski, M., and Hunt, E. *Providing Minimal-Sized Responses to DNS Queries That Have QTYPE=ANY*. Tech. rep. 8482. 2019. 10 pp. URL: https://rfc-editor.org/rfc/rfc8482.txt.

[2] Adamsky, F., Khayam, S. A., Jäger, R., and Rajarajan, M. P2P File-Sharing in Hell: Exploiting BitTorrent Vulnerabilities to Launch Distributed Reflective DoS Attacks. In: *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*. 2015. URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/p2p-file-sharing-hell-exploiting-bittorrent.

[3] Aitel, D. *SPIKE, a Fuzzer Creation Kit*. URL: http://www.immunitysec.com/downloads/SPIKE2.9.tgz (visited on 2021-06-01).

[4] *Anonymous Hacks US Government Site, Threatens Supreme 'Warheads'*. URL: https://mashable.com/archive/anonymous-hack-government-website-declares-war (visited on 2016-02-26).

[5] Anwar, R., Niaz, H., Choffnes, D., Cunha, Í., Gill, P., and Katz-Bassett, E. Investigating interdomain routing policies in the wild. In: *Proceedings of the 2015 Internet Measurement Conference*. 2015, 71–77.

[6] Arbor Networks. *Worldwide Infrastructure Security Report*. 2015. URL: https://www.arbornetworks.com/images/documents/WISR2016_EN_Web.pdf (visited on 2016-05-20).

[7] *AS65000 BGP Routing Table Analysis Report*. URL: https://bgp.potaroo.net/as2.0/ (visited on 2020-04-29).

[8] Atkins, D. and Austein, R. *Threat Analysis of the Domain Name System (DNS)*. Tech. rep. 3833. 2004. 16 pp. URL: https://rfc-editor.org/rfc/rfc3833.txt.

[9] *AWS said it mitigated a 2.3 Tbps DDoS attack, the largest ever*. URL: https://www.zdnet.com/article/aws-said-it-mitigated-a-2-3-tbps-ddos-attack-the-largest-ever/ (visited on 2020-06-18).

[10] Backes, M., Holz, T., Rossow, C., Rytilahti, T., Simeonovski, M., and Stock, B. On the Feasibility of TTL-based Filtering for DRDoS Mitigation. In: *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses*. 2016.

[11] Baker, F. and Savola, P. *Ingress Filtering for Multihomed Networks*. Tech. rep. 3704. 2004. 16 pp. URL: https://rfc-editor.org/rfc/rfc3704.txt.

[12] Barrett, C., Fontaine, P., and Tinelli, C. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2016. URL: www.smt-lib.org (visited on 2021-09-16).

[13] Basescu, C., Reischuk, R. M., Szalachowski, P., Perrig, A., Zhang, Y., Hsiao, H.-C., Kubota, A., and Urakawa, J. SIBRA: Scalable Internet Bandwidth Reservation Architecture. In: *NDSS '16*.

[14] Belenky, A. and Ansari, N. On deterministic packet marking. *Computer Networks* 10 (2007), 2677–2700.

[15] Bethencourt, J., Franklin, J., and Vernon, M. Mapping Internet Sensors with Probe Response Attacks. In: *Proceedings of the 14th Conference on USENIX Security Symposium*. 2005.

[16] Beverly, R. and Bauer, S. The Spoofer project: Inferring the extent of source address filtering on the Internet. In: *Usenix Sruti*. 2005, 53–59.

[17] Beverly, R., Koga, R., and Claffy, K. Initial longitudinal analysis of IP source spoofing capability on the Internet. *Internet Society* (2013), 313.

[18] Böhme, M., Pham, V.-T., and Roychoudhury, A. Coverage-based Greybox Fuzzing as Markov Chain. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 2016, 1032–1043. URL: https://doi.org/10.1145/2976749.2978428.

[19] Brenner, B. *RIPv1 Reflection DDoS Making a Comeback*. 2015. URL: https://blogs.akamai.com/2015/07/ripv1-reflection-ddos-making-a-comeback.html (visited on 2021-05-05).

[20] Bush, R., Maennel, O., Roughan, M., and Uhlig, S. Internet optometry: assessing the broken glasses in internet reachability. In: *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*. 2009, 242–253.

[21] Bush, R., Pelsser, C., Kuhne, M., Maennel, O., Mohapatra, P., Patel, K., and Evans, R. *RIPE Routing Working Group Recommendations on Route Flap Damping*. RIPE 580. 2013. URL: https://www.ripe.net/publications/docs/ripe-580 (visited on 2020-01-08).

[22] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 2008, 209–224. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[23] CAIDA. *The Spoofer Project*. URL: http://spoofer.cmand.org (visited on 2016-05-20).

[24] Chen, H., Li, Y., Chen, B., Xue, Y., and Liu, Y. FOT: a versatile, configurable, extensible fuzzing framework. In: *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 2018, 867–870. URL: https://doi.org/10.1145/3236024.3264593.

[25] Chen, P. and Chen, H. Angora: Efficient Fuzzing by Principled Search. en. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, 711–725. URL: https://ieeexplore.ieee.org/document/8418633/.

[26] Chen, R., Park, J.-M., and Marchany, R. A divide-and-conquer strategy for thwarting distributed denial-of-service attacks. *IEEE Transactions on Parallel and Distributed Systems* 5 (2007), 577–588.

[27] Christakis, M., Müller, P., and Wüstholz, V. Guiding dynamic symbolic execution toward unverified program executions. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 2016, 144–155. URL: https://doi.org/10.1145/2884781.2884843.

[28] *Cisco IOS IP Routing: BGP Command Reference*. URL: https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/iproute_bgp/command/irg-cr-book/bgp-a1.html#wp3674090369 (visited on 2020-01-08).

[29] Clayton, R. How Much Did Shutting Down McColo Help? *CEAS '09* ().

[30] Cloudflare. *RFC8482 - Saying goodbye to ANY*. 2019. URL: https://blog.cloudflare.com/rfc8482-saying-goodbye-to-any/ (visited on 2021-06-09).

[31] Cloudflare. *Memcached DDoS Attack*. URL: https://www.cloudflare.com/en-gb/learning/ddos/memcached-ddos-attack/ (visited on 2021-05-05).

[32] Colitti, L., Di Battista, G., Patrignani, M., Pizzonia, M., and Rimondini, M. Investigating prefix propagation through active BGP probing. *Microprocessors and Microsystems* 7 (2007), 460–474.

[33] Czyz, J., Kallitsis, M., Gharaibeh, M., Papadopoulos, C., Bailey, M., and Karir, M. Taming the 800 Pound Gorilla: The Rise and Decline of NTP DDoS Attacks. In: *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*. ACM. 2014, 435–448. URL: https://doi.org/10.1145/2663716.2663717.

[34] *DDoS extortionists target NZX, Moneygram, Braintree, and other financial services*. URL: https://www.zdnet.com/article/ddos-extortionists-target-nzx-moneygram-braintree-and-other-financial-services/ (visited on 2020-08-27).

[35] Dean, D., Franklin, M. K., and Stubblefield, A. An algebraic approach to IP traceback. *ACM Transactions on Information and System Security (TISSEC)* 2 (2002), 119–137.

[36] *Debian packages repo*. URL: https://www.debian.org/distrib/packages (visited on 2021-06-01).

[37] denandz. *FuzzoTron - A Fuzzing Harness built around OUSPG's Blab and Radamsa*. URL: https://github.com/denandz/fuzzotron (visited on 2021-06-01).

[38] Di Battista, G., Patrignani, M., and Pizzonia, M. Computing the types of the relationships between autonomous systems. In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*. IEEE. 2003, 156–165.

[39] Dimitropoulos, X., Krioukov, D., Fomenkov, M., Huffaker, B., Hyun, Y., Claffy, K., and Riley, G. AS relationships: Inference and validation. *ACM SIGCOMM Computer Communication Review* 1 (2007), 29–40.

[40] Dimitropoulos, X., Krioukov, D., Huffaker, B., Riley, G., et al. Inferring as relationships: Dead end or lively beginning? In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2005, 113–125.

[41] Dobbins, R. and Bjarnason, S. *Microsoft Remote Desktop Protocol (RDP) Reflection/Amplification DDoS Attack Mitigation Recommendations*. 2021. URL: https:

//www.netscout.com/blog/asert/microsoft-remote-desktop-protocol-rdp-reflectionamplification (visited on 2021-08-18).

[42] Doeppner, T. W., Klein, P. N., and Koyfman, A. Using router stamping to identify the source of IP packets. In: *Proceedings of the 7th ACM Conference on Computer and Communications Security*. 2000, 184–189.

[43] Dong, Q., Banerjee, S., Adler, M., and Hirata, K. Efficient probabilistic packet marking. In: *13TH IEEE International Conference on Network Protocols (ICNP'05)*. IEEE. 2005, 10–pp.

[44] Durumeric, Z., Wustrow, E., and Halderman, J. A. ZMap: Fast Internet-wide Scanning and Its Security Applications. In: *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*. 2013, 605–620. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/durumeric.

[45] Duwairi, B., Chakrabarti, A., and Manimaran, G. An efficient probabilistic packet marking scheme for IP traceback. In: *International Conference on Research in Networking*. Springer. 2004, 1263–1269.

[46] Erlebach, T., Hall, A., and Schank, T. Classifying customer-provider relationships in the Internet. *TIK-Report* (2002).

[47] Fan, R. and Chang, Y. Machine Learning for Black-Box Fuzzing of Network Protocols. In: *Information and Communications Security - 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings*. 2017, 621–632. URL: https://doi.org/10.1007/978-3-319-89500-0_53.

[48] Ferguson, P. and Senie, D. *Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*. Tech. rep. 2827. 2000. 10 pp. URL: https://rfc-editor.org/rfc/rfc2827.txt.

[49] Fiterau-Brostean, P., Jonsson, B., Merget, R., Ruiter, J. de, Sagonas, K., and Somorovsky, J. Analysis of DTLS Implementations Using Protocol State Fuzzing. In: *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*. 2020, 2523–2540. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[50] Fonseca, O. L. H. M., Cunha, Í., Fazzion, E. C., Jr., W. M., Junior, B., Ferreira, R. A., and Katz-Bassett, E. Tracking Down Sources of Spoofed IP Packets. In: *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*. IEEE. 2020, 208–216. URL: https://ieeexplore.ieee.org/document/9142717.

[51] Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., and Chen, Z. CollAFL: Path Sensitive Fuzzing. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 2018, 679–696. URL: https://doi.org/10.1109/SP.2018.00040.

[52] Ganesh, V., Leek, T., and Rinard, M. C. Taint-based directed whitebox fuzzing. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 2009, 474–484. URL: https://doi.org/10.1109/ICSE.2009.5070546.

[53] Gao, L. On inferring autonomous system relationships in the Internet. *IEEE/ACM Transactions on Networking (ToN)* 6 (2001), 733–745.

[54] Gao, Z. and Ansari, N. A practical and robust inter-domain marking scheme for IP traceback. *Computer Networks* 3 (2007), 732–750.

[55] Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., and Rieck, K. Pulsar: Stateful Black-Box Fuzzing of Proprietary Network Protocols. In: *Security and Privacy in Communication Networks - 11th International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Revised Selected Papers*. 2015, 330–347. URL: https://doi.org/10.1007/978-3-319-28865-9_18.

[56] Gasser, O., Scheitle, Q., Rudolph, B., Denis, C., Schricker, N., and Carle, G. The Amplification Threat Posed by PubliclyReachable BACnet Devices. *J. Cyber Secur. Mobil.* 1 (2017), 77–104. URL: https://doi.org/10.13052/jcsm2245-1439.614.

[57] Ge, X., Taneja, K., Xie, T., and Tillmann, N. DyTa: dynamic symbolic execution guided with static verification results. In: *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. 2011, 992–994. URL: https://doi.org/10.1145/1985793.1985971.

[58] Gilad, Y., Goberman, M., Herzberg, A., and Sudkovitch, M. CDN-on-Demand: An Affordable DDoS Defense via Untrusted Clouds. In: *Proceedings of NDSS 2016*. 2016.

[59] Giotsas, V., Luckie, M., Huffaker, B., and Claffy, K. Inferring complex AS relationships. In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. 2014, 23–30.

[60] Giotsas, V. and Zhou, S. Valley-free violation in Internet routing—Analysis based on BGP Community data. In: *2012 IEEE International Conference on Communications (ICC)*. IEEE. 2012, 1193–1197.

[61] GitLab. *GitLab Protocol Fuzzer*. URL: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce (visited on 2021-06-01).

[62] Godefroid, P., Levin, M. Y., and Molnar, D. A. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 3 (2012), 40–44. URL: https://doi.org/10.1145/2093548.2093564.

[63] Graham, R. D. *MASSCAN: Mass IP port scanner*. 2014. URL: https://github.com/robertdavidgraham/masscan (visited on 2016-05-12).

[64] Hedrick, C. *Routing Information Protocol*. Tech. rep. 1058. 1988. 33 pp. URL: https://rfc-editor.org/rfc/rfc1058.txt.

[65] Hummel, B. and Kosub, S. Acyclic type-of-relationship problems on the internet: an experimental analysis. In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. 2007, 221–226.

[66] Hyun, Y., Broido, A., et al. *Traceroute and BGP AS path incongruities*. Tech. rep. Cooperative Association for Internet Data Analysis (CAIDA), 2003.

[67] *IP to ASN mapping*. URL: https://www.team-cymru.org/IP-ASN-mapping.html (visited on 2016-05-20).

[68] *ISP Cut off From Internet After Security Concerns*. URL: https://www.pcworld.com/article/153734/mccolo_isp_security.html (visited on 2021-02-03).

[69] Jerkovich, D. *rage against the network*. URL: https://github.com/deanjerkovich/rage_fuzzer (visited on 2021-06-01).

[70] John, A. and Sivakumar, T. Ddos: Survey of traceback methods. *International Journal of Recent Trends in Engineering* 2 (2009), 241.

[71] Karami, M. and McCoy, D. Understanding the emerging threat of ddos-as-a-service. In: *Presented as part of the 6th {USENIX} Workshop on Large-Scale Exploits and Emergent Threats*. 2013.

[72] Karami, M., Park, Y., and McCoy, D. Stress testing the booters: Understanding and undermining the business of DDoS services. In: *Proceedings of the 25th International Conference on World Wide Web*. 2016, 1033–1043.

[73] Katz-Bassett, E., Choffnes, D. R., Cunha, Í., Scott, C., Anderson, T., and Krishnamurthy, A. Machiavellian routing: improving internet availability with BGP poisoning. In: *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*. 2011, 1–6.

[74] Katz-Bassett, E., Scott, C., Choffnes, D. R., Cunha, Í., Valancius, V., Feamster, N., Madhyastha, H. V., Anderson, T., and Krishnamurthy, A. LIFEGUARD: Practical repair of persistent route failures. *ACM SIGCOMM Computer Communication Review* 4 (2012), 395–406.

[75] Korkmaz, T., Gong, C., Sarac, K., and Dykes, S. G. Single packet IP traceback in AS-level partial deployment scenario. *International Journal of Security and Networks* 1-2 (2007), 95–108.

[76] Krebs, B. *DDoS on Dyn Impacts Twitter, Spotify, Reddit*. 2016. URL: https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/ (visited on 2020-06-18).

[77] Krebs, B. *DDoS Mitigation Firm Founder Admits to DDoS*. 2020. URL: https://krebsonsecurity.com/2020/01/ddos-mitigation-firm-founder-admits-to-ddos/ (visited on 2021-08-06).

[78] Kreibich, C., Warfield, A., Crowcroft, J., Hand, S., and Pratt, I. Using Packet Symmetry to Curtail Malicious Traffic. In: *Proceedings of the 4th Workshop on Hot Topics in Networks (Hotnets-VI)*. 2005.

[79] Kührer, M., Hupperich, T., Bushart, J., Rossow, C., and Holz, T. Going Wild: Large-Scale Classification of Open DNS Resolvers. In: *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015*. 2015, 355–368. URL: https://doi.org/10.1145/2815675.2815683.

[80] Kührer, M., Hupperich, T., Rossow, C., and Holz, T. Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In: *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 2014, 111–125. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kuhrer.

[81] Kührer, M., Hupperich, T., Rossow, C., and Holz, T. Hell of a Handshake: Abusing TCP for Reflective Amplification DDoS Attacks. In: *8th USENIX Workshop on Offensive Technologies, WOOT '14, San Diego, CA, USA, August 19, 2014*. 2014. URL: https://www.usenix.org/conference/woot14/workshop-program/presentation/kuhrer.

[82] Labovitz, C., Ahuja, A., Bose, A., and Jahanian, F. Delayed Internet routing convergence. *ACM SIGCOMM Computer Communication Review* 4 (2000), 175–187.

[83] Lemieux, C., Padhye, R., Sen, K., and Song, D. PerfFuzz: automatically generating pathological inputs. en. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2018, 254–265. URL: https://dl.acm.org/doi/10.1145/3213846.3213874.

[84] Lepinski, M. and Kent, S. *An Infrastructure to Support Secure Internet Routing*. Tech. rep. 6480. 2012. 24 pp. URL: https://rfc-editor.org/rfc/rfc6480.txt.

[85] Lepinski, M., Kent, S., and Kong, D. *A Profile for Route Origin Authorizations (ROAs)*. Tech. rep. 6482. 2012. 9 pp. URL: https://rfc-editor.org/rfc/rfc6482.txt.

[86] Li, F., Durumeric, Z., Czyz, J., Karami, M., Bailey, M., McCoy, D., Savage, S., and Paxson, V. You've Got Vulnerability: Exploring Effective Vulnerability Notifications. In: *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. 2016, 1033–1050. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/li.

[87] Li, J., Sung, M., Xu, J., and Li, L. Large-scale IP traceback in high-speed Internet: Practical techniques and theoretical foundation. In: *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*. IEEE. 2004, 115–129.

[88] Liu, B., Li, J., Wei, T., Berg, S., Ye, J., Li, C., Zhang, C., Zhang, J., and Han, X. SF-DRDoS: The store-and-flood distributed reflective denial of service attack. *Comput. Commun.* (2015), 107–115. URL: https://doi.org/10.1016/j.comcom.2015.06.008.

[89] Luckie, M., Beverly, R., Koga, R., Keys, K., Kroll, J. A., and, k. claffy k. Network Hygiene, Incentives, and Regulation: Deployment of Source Address Validation in the Internet. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, 465–480.

[90] Luckie, M., Huffaker, B., Dhamdhere, A., Giotsas, V., et al. AS relationships, customer cones, and validation. In: *Proceedings of the 2013 conference on Internet measurement conference*. ACM. 2013, 243–256.

[91] MacFarland, D. C., Shue, C. A., and Kalafut, A. J. The best bang for the byte: Characterizing the potential of DNS amplification attacks. *Comput. Networks* (2017), 12–21. URL: https://doi.org/10.1016/j.comnet.2017.02.007.

[92] Mao, Z. M., Govindan, R., Varghese, G., and Katz, R. H. Route flap damping exacerbates Internet routing convergence. In: *ACM SIGCOMM Computer Communication Review*. 4. ACM. 2002, 221–233.

[93] *MasterCard goes down as Anonymous launch 2nd attack*. URL: https://news.netcraft.com/archives/2010/12/12/mastercard-goes-down-as-anonymous-launch-2nd-attack.html (visited on 2010-12-12).

[94] MaxMind. *GeoLite2 Free Downloadable Databases*. URL: https://dev.maxmind.com/geoip/geoip2/geolite2/ (visited on 2016-05-20).

[95] McMinn, P. Search-based software test data generation: a survey. *Softw. Test. Verification Reliab.* 2 (2004), 105–156. URL: https://doi.org/10.1002/stvr.294.

[96] Mills, D. L. *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. Tech. rep. 1305. 1992. 109 pp. URL: https://rfc-editor.org/rfc/rfc1305.txt.

[97] Moon, S.-J., Yin, Y., Sharma, R. A., Yuan, Y., Spring, J. M., and Sekar, V. Accurately Measuring Global Risk of Amplification Attacks using AmpMap. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, 3881–3898. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/moon.

[98] Mühlbauer, W., Feldmann, A., Maennel, O., Roughan, M., and Uhlig, S. Building an AS-topology model that captures route diversity. *ACM SIGCOMM Computer Communication Review* 4 (2006), 195–206.

[99] *NETSCOUT Arbor Confirms 1.7 Tbps DDoS Attack; The Terabit Attack Era Is Upon Us*. URL: https://www.netscout.com/blog/asert/netscout-arbor-confirms-17-tbps-ddos-attack-terabit-attack-era (visited on 2020-06-18).

[100] *Netscout threat intelligence report - DDoS in a Time of Pandemic*. URL: https://www.netscout.com/sites/default/files/2021-04/ThreatReport_2H2020_FINAL_0.pdf (visited on 2021-08-05).

[101] Open Reverse Code Engineering. *Sulley: A pure-python fully automated and unattended fuzzing framework*. URL: https://github.com/OpenRCE/sulley (visited on 2021-06-01).

[102] Österlund, S., Razavi, K., Bos, H., and Giuffrida, C. ParmeSan: Sanitizer-guided Greybox Fuzzing. en. In: 2020, 2289–2306. URL: https://www.usenix.org/system/files/sec20-osterlund.pdf.

[103] Padhye, R., Lemieux, C., Sen, K., Simon, L., and Vijayakumar, H. FuzzFactory: domain-specific fuzzing with waypoints. en. *Proceedings of the ACM on Programming Languages* OOPSLA (2019), 1–29. URL: https://dl.acm.org/doi/10.1145/3360600.

[104] Paxson, V. An analysis of using reflectors for distributed denial-of-service attacks. *Comput. Commun. Rev.* 3 (2001), 38–47. URL: https://doi.org/10.1145/505659.505664.

[105] Pelsser, C., Bush, R., Patel, K., Mohapatra, P., and Maennel, O. *Making Route Flap Damping Usable*. Tech. rep. 7196. 2014. 8 pp. URL: https://rfc-editor.org/rfc/rfc7196.txt.

[106] Pelsser, C., Maennel, O., Mohapatra, P., Bush, R., and Patel, K. Route flap damping made usable. In: *International Conference on Passive and Active Network Measurement*. Springer. 2011, 143–152.

[107] Pereyda, J. *boofuzz: Network Protocol Fuzzing for Humans*. URL: https://github.com/jtpereyda/boofuzz (visited on 2021-06-01).

[108] Perrig, A., Song, D., and Yaar, A. *StackPi: A New Defense Mechanism against IP Spoofing and DDoS Attacks*. Tech. rep. 2003.

[109] Petsios, T., Zhao, J., Keromytis, A. D., and Jana, S. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. en. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, 2155–2168. URL: https://dl.acm.org/doi/10.1145/3133956.3134073.

[110] Pham, V.-T., Böhme, M., and Roychoudhury, A. AFLNET: A Greybox Fuzzer for Network Protocols. In: *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. 2020, 460–465. URL: https://doi.org/10.1109/ICST46399.2020.00062.

[111] Poeplau, S. and Francillon, A. Symbolic execution with SymCC: Don't interpret, compile! In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, 181–198. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[112] Prince, M. *The DDoS That Almost Broke the Internet*. 2013. URL: https://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet/ (visited on 2016-05-20).

[113] Prince, M. *The DDoS That Knocked Spamhaus Offline (And How We Mitigated It)*. 2013. URL: https://blog.cloudflare.com/the-ddos-that-knocked-spamhaus-offline-and-ho/ (visited on 2020-06-18).

[114] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In: *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March*

1, 2017. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

[115] Rekhter, Y., Li, T., and Hares, S. *A Border Gateway Protocol 4 (BGP-4)*. Tech. rep. 4271. 2006. 104 pp. URL: https://rfc-editor.org/rfc/rfc4271.txt.

[116] Rijswijk-Deij, R. van, Sperotto, A., and Pras, A. DNSSEC and its potential for DDoS attacks: a comprehensive measurement study. In: *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*. ACM. 2014, 449–460. URL: https://doi.org/10.1145/2663716.2663731.

[117] *RIPE Atlas*. https://atlas.ripe.net. URL: https://atlas.ripe.net (visited on 2019-12-03).

[118] *RIPE Routing Information Service (RIS)*. URL: http://www.ripe.net/data-tools/stats/ris (visited on 2020-11-02).

[119] *RIPEstat Data API*. URL: https://stat.ripe.net/docs/data_api (visited on 2020-01-21).

[120] Rossow, C. Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In: *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. 2014. URL: https://www.ndss-symposium.org/ndss2014/amplification-hell-revisiting-network-protocols-ddos-abuse.

[121] Rutishauser, D. *Fuzzing For Worms*. URL: https://github.com/dobin/ffw (visited on 2021-06-01).

[122] Sanfilippo, S. *New TCP Scan Method*. URL: http://seclists.org/bugtraq/1998/Dec/79 (visited on 2016-05-22).

[123] Santanna, J., Durban, R., Sperotto, A., and Pras, A. Inside Booters: An Analysis on Operational Databases. In: *14th IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2015.

[124] Santanna, J. J., Rijswijk-Deij, R. van, Hofstede, R., Sperotto, A., Wierbosch, M., Granville, L. Z., and Pras, A. Booters—An analysis of DDoS-as-a-service attacks. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, 243–251.

[125] Sargent, M., Kristoff, J., Paxson, V., and Allman, M. On the Potential Abuse of IGMP. *Comput. Commun. Rev.* 1 (2017), 27–35. URL: https://doi.org/10.1145/3041027.3041031.

[126] Savage, S., Wetherall, D., Karlin, A., and Anderson, T. Network support for IP traceback. *IEEE/ACM transactions on networking* 3 (2001), 226–237.

[127] Savage, S., Wetherall, D., Karlin, A. R., and Anderson, T. E. Practical Network Support for IP Traceback. In: *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 4. ACM. 2000, 295–306.

[128] Schlinker, B., Arnold, T., Cunha, I., and Katz-Bassett, E. PEERING: Virtualizing BGP at the Edge for Research. In: *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*. 2019, 51–67.

[129] Schuchard, M., Geddes, J., Thompson, C., and Hopper, N. Routing around decoys. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, 85–96.

[130] Schuchard, M., Mohaisen, A., Foo Kune, D., Hopper, N., Kim, Y., and Vasserman, E. Y. Losing control of the internet: using the data plane to attack the control plane. In:

Proceedings of the 17th ACM conference on Computer and communications security. 2010, 726–728.

[131] *SELinux Project*. URL: https://github.com/SELinuxProject (visited on 2021-06-01).

[132] Shokri, R., Varshovi, A., Mohammadi, H., Yazdani, N., and Sadeghian, B. DDPM: dynamic deterministic packet marking for IP traceback. In: *2006 14th IEEE International Conference on Networks*. IEEE. 2006, 1–6.

[133] Sidiroglou-Douskos, S., Lahtinen, E., Rittenhouse, N., Piselli, P., Long, F., Kim, D., and Rinard, M. C. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*. 2015, 473–486. URL: https://doi.org/10.1145/2694344.2694389.

[134] Smith, J. M., Birkeland, K., McDaniel, T., and Schuchard, M. Withdrawing the BGP Re-Routing Curtain. In: *Network and Distributed System Security Symposium (NDSS)*. 2020.

[135] Smith, J. M. and Schuchard, M. Routing around congestion: Defeating DDoS attacks and adverse network conditions via reactive BGP routing. In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, 599–617.

[136] Smith, P. and Panigl, C. *RIPE Routing Working Group Recommendations On Route-flap Damping*. RIPE 378. 2006. URL: https://www.ripe.net/publications/docs/ripe-378 (visited on 2020-01-08).

[137] Snoeren, A. C., Partridge, C., Sanchez, L. A., Jones, C. E., Tchakountio, F., Kent, S. T., and Strayer, W. T. Hash-based IP traceback. *ACM SIGCOMM Computer Communication Review* 4 (2001), 3–14.

[138] Snoeren, A. C., Partridge, C., Sanchez, L. A., Jones, C. E., Tchakountio, F., Schwartz, B., Kent, S. T., and Strayer, W. T. Single-packet IP traceback. *IEEE/ACM Transactions on networking* 6 (2002), 721–734.

[139] Song, C., Yu, B., Zhou, X., and Yang, Q. SPFuzz: A Hierarchical Scheduling Framework for Stateful Network Protocol Fuzzing. *IEEE Access* (2019), 18490–18499. URL: https://doi.org/10.1109/ACCESS.2019.2895025.

[140] Song, D. X. and Perrig, A. Advanced and authenticated marking schemes for IP traceback. In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. IEEE. 2001, 878–886.

[141] Sriram, K., Montgomery, D., and Haas, J. *Enhanced Feasible-Path Unicast Reverse Path Forwarding*. Tech. rep. 8704. 2020. 17 pp. URL: https://rfc-editor.org/rfc/rfc8704.txt.

[142] Stadje, W. The Collector's Problem with Group Drawings. *Advances in Applied Probability* 4 (1990). URL: http://www.jstor.org/stable/1427566.

[143] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. en. In: *Proceedings 2016 Network and Distributed System Security Symposium*. 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf.

[144] Subramanian, L., Agarwal, S., Rexford, J., and Katz, R. H. Characterizing the Internet hierarchy from multiple vantage points. In: *Proceedings. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE. 2002, 618–627.

[145] Sun, X., Torres, R., and Rao, S. DDoS Attacks by Subverting Membership Management in P2P systems. In: *Proceedings of the 3rd IEEE Workshop on Secure Network Protocols (NPSec)*. 2007.

[146] Sun, X., Torres, R., and Rao, S. On the Feasibility of Exploiting P2P Systems to Launch DDoS Attacks. In: *Journal of Peer-to-Peer Networking and Applications*. 1. 2010.

[147] Sung, M., Xu, J., Li, J., and Li, L. Large-scale IP traceback in high-speed internet: practical techniques and information-theoretic foundation. *IEEE/ACM Transactions on Networking* 6 (2008), 1253–1266.

[148] Talos, C. *Mutiny Fuzzing Framework*. URL: https://github.com/Cisco-Talos/mutiny-fuzzer (visited on 2021-06-01).

[149] *Team Cymru IP to ASN Lookup v1.0*. URL: https://whois.cymru.com/ (visited on 2019-12-03).

[150] *Telegram says 'whopper' DDoS attack launched mostly from China*. URL: https://www.zdnet.com/article/telegram-says-whopper-ddos-attack-launched-mostly-from-china/ (visited on 2019-06-13).

[151] *The CAIDA AS Relationships Dataset, Jun 2019 - Jan 2020*. URL: http://www.caida.org/data/active/as-relationships/ (visited on 2019-06-01/2020-02-10).

[152] *The search engine for the Internet of Things*. URL: https://www.shodan.io/ (visited on 2020-06-18).

[153] Thomas, D. R., Clayton, R., and Beresford, A. R. 1000 days of UDP amplification DDoS attacks. In: *2017 APWG Symposium on Electronic Crime Research, eCrime 2017, Phoenix, AZ, USA, April 25-27, 2017*. IEEE. 2017, 79–84. URL: https://doi.org/10.1109/ECRIME.2017.7945057.

[154] Tran, M., Choi, I., Moon, G. J., Vu, A. V., and Kang, M. S. A Stealthier Partitioning Attack against Bitcoin Peer-to-Peer Network. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020, 894–909. URL: https://doi.org/10.1109/SP40000.2020.00027.

[155] Tran, M., Kang, M. S., Hsiao, H.-C., Chiang, W.-H., Tung, S.-P., and Wang, Y.-S. On the Feasibility of Rerouting-based DDoS Defenses. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, 1169–1184.

[156] *University of Oregon Route Views Archive Project*. URL: http://routeviews.org/ (visited on 2020-11-02).

[157] Villamizar, C., Chandra, R., and Govindan, D. R. *BGP Route Flap Damping*. Tech. rep. 2439. 1998. 37 pp. URL: https://rfc-editor.org/rfc/rfc2439.txt.

[158] Vixie, P. *Response Rate Limiting in the Domain Name System (DNS RRL)*. 2012. URL: http://www.redbarn.org/dns/ratelimits (visited on 2021-05-05).

[159] Wang, A., Mohaisen, A., Chang, W., and Chen, S. Capturing DDoS Attack Dynamics Behind the Scenes. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2015.

[160] Wang, T., Wei, T., Gu, G., and Zou, W. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In: *31st IEEE Sym-*

posium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. 2010, 497–512. URL: https://doi.org/10.1109/SP.2010.37.

[161] Wang, X. and Reiter, M. K. Mitigating Bandwidth-Exhaustion Attacks Using Congestion Puzzles. In: *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. 2004.

[162] *Whole Program LLVM*. Version 1.2.8. URL: https://github.com/SRI-CSL/whole-program-llvm (visited on 2021-04-30).

[163] Xia, J. and Gao, L. On the evaluation of AS relationship inferences [Internet reachability/traffic flow applications]. In: *IEEE Global Telecommunications Conference, 2004. GLOBECOM'04.* IEEE. 2004, 1373–1377.

[164] Yaar, A., Perrig, A., and Song, D. Pi: A path identification mechanism to defend against DDoS attacks. In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2003, 93–107.

[165] Yaar, A., Perrig, A., and Song, D. StackPi: New packet marking and filtering mechanisms for DDoS and IP spoofing defense. *IEEE Journal on Selected Areas in Communications* 10 (2006), 1853–1863.

[166] Yaar, A., Perrig, A., and Song, D. X. Pi: A Path Identification Mechanism to Defend against DDoS Attack. In: *IEEE S&P '03*.

[167] Yang, X. and kenshin. *CLDAP is Now the No.3 Reflection Amplified DDoS Attack Vector, Surpassing SSDP and CharGen*. 2017. URL: https://blog.netlab.360.com/cldap-is-now-the-3rd-reflection-amplified-ddos-attack-vector-surpassing-ssdp-and-chargen-en/ (visited on 2021-08-18).

[168] Zalewski, M. *American Fuzzy Lop: a security-oriented fuzzer*. 2010. URL: http://lcamtuf.coredump.cx/afl/ (visited on 2021-05-01).

[169] Zhang, X., Hsiao, H.-C., Hasker, G., Chan, H., Perrig, A., and Andersen, D. G. SCION: Scalability, Control, and Isolation on Next-Generation Networks. In: *IEEE S&P '11*.

[170] Zhang, Y., Oliveira, R., Zhang, H., and Zhang, L. Quantifying the pitfalls of traceroute in AS connectivity inference. In: *International Conference on Passive and Active Network Measurement*. Springer. 2010, 91–100.