AMPFUZZ: Fuzzing for Amplification DDoS Vulnerabilities

Johannes Krupp
CISPA Helmholtz Center for Information Security

Ilya Grishchenko[⊛]
University of California, Santa Barbara

Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

Amplification DDoS attacks remain a prevalent and severe threat to the Internet, with recent attacks reaching the Tbps range. However, all amplification attack vectors known to date were either found by researchers through laborious manual analysis or could only be identified *postmortem* following large attacks. Ideally, though, an attack vector is discovered and mitigated before the first attack can occur.

To this end, we present AMPFUZZ, the first systematic approach to finding amplification vectors in UDP services in a protocol-agnostic way. AMPFUZZ is based on the state-of-the-art greybox fuzzing boosted by a novel technique to make fuzzing UDP-aware, which significantly increases performance. We evaluate AMPFUZZ on 28 Debian network services, where we (re-)discover 7 known and 6 previously unreported amplification vulnerabilities.

1 Introduction

For many years, amplification Distributed Denial-of-Service (DDoS) attacks [74] constitute the most powerful volumetric DDoS strategy. These attacks abuse the fact that several UDP-based network services do not (or cannot, without changing decades-old protocols) verify client IP addresses. An attacker can turn such vulnerable services into attack traffic amplifiers by sending them forged requests under their victim's identity. Being unable to verify the request origin, these services unwittingly flood the victim with their (unsolicited) answers. Past attacks temporarily disrupted core Internet services, such as Paypal, Spotify, Twitter, Reddit, and eBay, with recent attacks in 2021 peaking at more than 2.4 Tbps attack bandwidth [22, 26, 45, 60].

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 massive potential [23]. Similarly, the discovery of amplification vectors in RIPv1 [4] has quickly led to a stark increase in amplification abuses of this protocol [17]. Surprisingly, while there are automated approaches to finding variants of known vulnerabilities and estimating their amplification potential [59], we lack any automation in discovering *new* amplification vectors. 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 [62]. 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 safeguarding 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 or large-scale disclosure operations have massively reduced the number of vulnerable services [50, 85]. Second, early knowledge of vulnerabilities allows monitoring active exploitation in amplification attacks with the help of amplification DDoS honeypots [44, 82]. Third, anti-DDoS services can ingest novel attack vectors in automated defense systems that filter attack traffic *before* 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 of 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

[⊛] The author contributed while being employed at CISPA.

in which it can react to requests, and (ii) when a service has finished processing a request, both of which are not readily observable to the outside world. Finally, amplification vulnerabilities have an inherent notion of severity: their utility to an attacker is quantified by the maximal request-response ratio. This makes them different from other bug classes. While for program crashes we are only interested in whether a certain request type *does* trigger a crash or not, in the case of amplification vulnerabilities we are further interested in maximizing the amplification per request type.

In this paper, we provide the first principled, protocol-agnostic approach to revealing new amplification vulnerabilities in network services. We propose a greybox fuzzer that aims to discover network requests that result in significantly larger responses. We overcome the challenges above by using a directed fuzzer that prioritizes paths leading to sending functions (e.g., `sendto`), which we extend with UDP-awareness. To this end, we combine static analysis with lightweight instrumentation to notify the fuzzer when the program expects input. In many cases, these techniques also enable us to proactively terminate fuzzing executions that generate no output instead of relying on expensive timeouts. Finally, two simple yet effective mutation strategies maximize the amplification factor of potential vulnerabilities.

As another application of AMPFUZZ, we extend its pipeline to provide the necessary means to *monitor* amplification abuse in an automated way through DDoS honeypots. Until now, amplification DDoS honeypots solely rely on experts who have to craft protocol-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 automate the entire process from scanning a service for amplification vulnerabilities to creating honeypots that emulate the discovered behavior.

Summarizing, we provide the following contributions:

- We present AMPFUZZ, the first systematic and protocol-agnostic approach to discover amplification vulnerabilities in network services based on directed greybox fuzzing.
- We introduce the novel concept of UDP-awareness providing a significant improvement in the fuzzing performance of AMPFUZZ.
- We evaluate our open-source[1] implementation of AMPFUZZ on 28 services from the Debian repositories. AMPFUZZ identifies 19 implementations as vulnerable, revealing 6 previously undetected amplification vectors.

2 Background

We first briefly introduce the general concepts of fuzzing, amplification DDoS attacks, and amplification vulnerabilities.

2.1 Fuzzing

Fuzzing has become a popular technique to find software vulnerabilities [57]. 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 applied to finding performance issues [52, 69] or unexpectedly large memory allocations [65].

Blackbox fuzzers generate inputs purely at random and are oblivious to the SuT's inner workings. This allows them to test many inputs quickly, but generally fails on programs that require structured input. Contrarily, whitebox fuzzers [18, 38] 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 using a constraint solver. This enables whitebox fuzzing also to 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 is (guided) greybox fuzzing [16, 20, 32, 58] aiming at the sweet spot between the strategies above. 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 [20, 72, 87] or reaching specific points in the program [19, 21, 33, 36, 78, 86, 89].

2.2 Amplification Attacks

Amplification attacks [74] are the predominant type of volumetric Distributed Denial-of-Service (DDoS) attacks, with tens of thousands of amplification attacks *per day* continuously threatening the availability of essential network services [44]. As depicted in Figure 1, in such an attack, to attack their victims, the attackers abuse amplifiers, i.e., innocent services that suffer from amplification vulnerabilities. In particular, these amplifiers will send our large responses without verifying the requests' source addresses. The attacker will hence craft requests carrying the victim's address as the claimed source address. In turn, the vulnerable services will send their responses with no ill intent towards the victim, thus flooding it with unsolicited traffic. By focusing on services with a substantial response-request ratio, the attack bandwidth

[1] The code of AMPFUZZ, along with all the evaluation artifacts, are publicly available at `https://github.com/amp-fuzz/AmpFuzz`

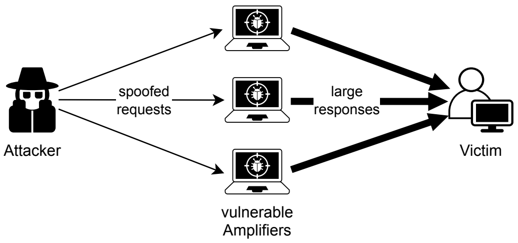


Figure 1: Amplification attack

arriving at the victim can be orders of magnitude larger than the bandwidth that has to be invested by the attacker.

2.3 Amplification Vulnerabilities

Barring a few exceptions [15, 51] all known amplification vulnerabilities are located in UDP-based protocols, including widely-used ones such as DNS, NTP, or SNMP [74]. Fortunately, once discovered, these vulnerabilities can be mitigated. For instance, the `monlist` debugging feature in the Network Time Protocol (NTP) [5] allowed amplifying traffic by up to $4{,}670\times$. After its discovery, Kührer et al. coordinated vulnerability disclosure, leading to a reduction of the vulnerable systems by 92% within just 10 weeks [50]. Similarly, the amplification potential in DNS was significantly mitigated by widescale deployment of rate limiting and message truncation [85], or by disabling non-critical features [8, 24].

However, to date, we lack a systematic way of finding amplification vectors. Dissimilar to other security vulnerabilities like memory corruption errors, they do not cause crashes or lead to anomalous software behavior. Instead, most amplification vectors rely on *intended* protocol or implementation features. As such, they resemble an entirely new bug class that fuzzers and other analysis tools have never been applied to and which come with unique challenges.

3 Fuzzing for Amplification

In this paper, we thus aim to adopt fuzzing to the domain of amplification vulnerabilities. Specifically, our goal is to find reasonably small UDP requests that trigger larger (amplifying) responses from a given network daemon.

3.1 Amplification Fuzzing Challenges

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

Lack of Protocol Knowledge Network protocol fuzzing is challenging *per se*, as each protocol comes with its own formats, syntax, and features. While past research on network fuzzing has focused on generative approaches that provide the fuzzer with a protocol specification from which it attempts to generate requests, this approach has severe limitations. Firstly, it requires knowledge and a formal description of the target's protocol. More importantly, though, it restricts the fuzzer to requests closely matching the said protocol. Yet many of the known amplification vectors rely on either custom, implementation-specific extensions (e.g., NTP `monlist` [6]) or exploit the target's handling of malformed requests (e.g., WS-Discovery [73]). To find such cases, we thus do not want to assume any *a priori* knowledge of the fuzzing targets.

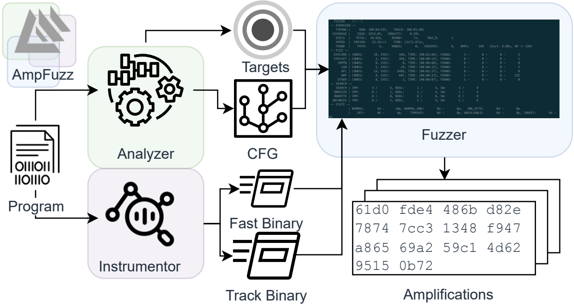
Lack of UDP State UDP network daemons are further challenging targets for fuzzing, as it is non-obvious when the daemon under test has finished processing a request. They will often silently discard invalid requests, providing no feedback to the fuzzer. And even if the network daemons respond to the fuzzer, it remains unclear whether further packets will follow. A similar situation occurs during the daemon startup: At which point is it ready to accept requests from the fuzzer? A request sent too early will be dropped by the network layer, resulting in an ICMP unreachable packet at best. 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 timeouts can address both startup and response delays—the *de facto* workaround used in the literature, e.g., [30]—this solution is suboptimal, as the actual processing time depends on both the target and the current request. Static timeouts will thus unnecessarily slow down fuzzing for some targets while terminating others prematurely.

Unexplored Vulnerability Class Lastly, amplification vulnerabilities inherently differ from other classical bug classes. While a request triggering a program crash unequivocally indicates a bug, not every request leading to a response constitutes an amplification vector. Instead, whether or not a request can be leveraged for amplification depends on the request-response size ratio. To accurately identify a target's amplification potential, the fuzzer must thus also be able to explore request variants to maximize this ratio—by either decreasing the request or increasing the response size.

3.2 Design Overview

We tackle the challenges mentioned above with AmpFuzz, as shown in Figure 2. We base AmpFuzz on the overall fuzzing pipeline of ParmeSan [89], a *directed fuzzing* extension of the mutation-based greybox fuzzer Angora [20], which consists of three main components: a static analyzer, a program instrumentor, and the actual fuzzer.

In a pre-processing step, the *static analyzer* extracts a control-flow graph (CFG) and a list of interesting target lo-



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

Figure 2: AmpFuzz outline

cations to guide the fuzzer to. The *program instrumentor* generates two instrumented versions of the service under test. The first one, including only a lightweight coverage collection, allows the fuzzer to quickly test whether a new input reaches new and undiscovered parts of the program. The second version, yet considerably slower version including a dataflow analysis framework, can then be run on only those inputs selectively to provide additional information about which input bytes are relevant for branching decisions. The *fuzzer* finally 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.

3.3 Protocol-Agnostic Fuzzing

To fuzz a service for amplification vulnerabilities *without* a specification of its protocol, we build on the directed fuzzing capabilities of ParmeSan [89] and the dataflow analysis of Angora [20]. Specifically, we collect all calls to network functions that send out packets as targets during the static analysis phase. By guiding the fuzzer to these locations, we directly focus on requests that trigger responses. Moreover, we extend the dataflow analysis to recognize network functions that receive packets as taint sources. Together with the byte-level taint-tracking, the fuzzer can perform targeted mutations of the inputs, eventually producing requests that are “valid” enough to generate responses.

3.4 UDP-Aware Fuzzing

To help AMPFUZZ decide when the target is ready to accept requests and when the processing of a request has concluded, we add additional lightweight instrumentation to the SuT, which aims to make these events observable to the fuzzer. To this end, we classify network functions into three groups, as shown in Table 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.

Table 1: Network function classification

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

Beginning of Request Processing We can ascertain that the SuT is ready to accept requests whenever it attempts to read or wait for a packet from the network, i.e., whenever a function from the *source* or *blocking* category is called on a UDP socket bound to the current fuzzing port. Thus, we can communicate this to the fuzzer (e.g., through a shared memory semaphore) by hooking all calls to functions from those categories and inspecting the state of the passed socket. As a socket’s listening state and port can be obtained at runtime, no extra socket accounting mechanism is required.

End of Request Processing Unfortunately, determining when request handling is completed is not as straightforward as it requires reasoning about the SuT’s *future* execution traces. Essentially we need to answer the question "Can this program still reach a *sink* without calling a *source* or *blocking*?". For multithreaded programs, in particular, this is a non-trivial property.

However, we can build a partial solution by considering each thread of a program individually: If, after a request has been received, the current *thread* is blocked at a *source* or *blocking* function called on the fuzzing socket, then this thread cannot yield a response to the original request. In those cases, we can safely terminate the current thread instead. Terminating only the current thread but not the entire SuT ensures that request handling in other threads can still proceed. Yet, as long as all threads eventually exit independently or can be early terminated, the fuzzer can observe that the entire SuT process has finished and proceed with the next round.

We can again implement this approach by hooking all calls to *source* and *blocking* functions and inspecting the socket argument. However, we can also further aid this approach with a simple static analysis and instrumentation that injects additional “check-and-terminate” calls into the program. This is particularly helpful in cases where the SuT still performs expensive computations or I/O operations before accepting the subsequent request, like adding lines to a logfile. To find these “check-and-terminate” edges, we first need to establish which (strict) basic blocks can reach a *sink* function before a *source* function. We will call those basic blocks *sink-capable*. Specifically, *sink-capable* 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 sink-

capable successor (e.g., `if` node in `handle` and node with a call to `handle` in `main`).

We can compute the set of *sink-capable* basic blocks using a fixed-point iteration pass over the targets' inter-procedural CFG. Undecided basic blocks after this fixed-point iteration are *non-sink-capable*. 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 SuT prematurely in the presence of dynamic calls or calls to shared libraries, we overapproximate function calls that cannot be analyzed statically as *sink-capable*. We can then add our check-and-terminate functionality to all edges leading from a *sink-capable* basic block to a *non-sink-capable* basic block.

Figure 3 shows this approach on an example service following a structure commonly observed in network daemons: After performing some initialization (`init`), e.g., obtaining a socket, the `main` function enters a `while`-loop that continuously waits for incoming packets (*source* `recv`) and invokes a handler function `handle` on every request. The handler `handle` checks the request (`if`) and either proceeds with a reply (*sink* `send`) or goes straight to logging the request (`log_request`) before returning (`return`). Following the rules above, we inject additional check-and-terminate calls into those edges marked in red (→).

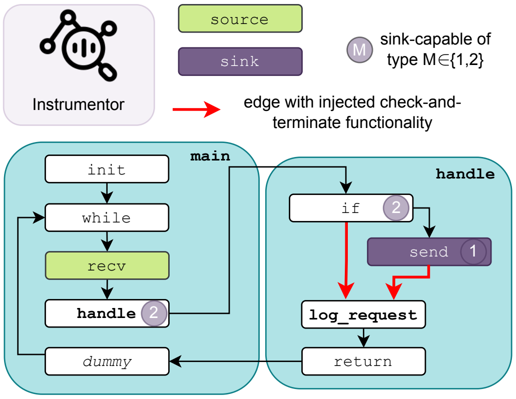


Figure 3: High-level example of our static analysis execution

3.5 Amplification Feedback and Optimization

Our protocol-agnostic and UDP-aware fuzzing measures enable AMPFUZZ to search for requests that generate responses efficiently. To focus on *amplification vectors*, we must further enable the fuzzer to maximize the ratio between request and response sizes. To this end, we build upon the *bandwidth amplification factor* (BAF) as defined by Rossow [74]:

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

However, to correctly handle services that reply to zero-length packets or 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, and take the sum of all response packets:

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

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

In classical mutation-based fuzzing, fuzz inputs are recorded as new seeds for a subsequent mutation if they increase coverage, i.e., exercise new paths of the program. For AMPFUZZ, we further record inputs that increase the amplification factor, globally or locally for their path. We explicitly include inputs with a $BAF_{L2} \leq 1$ (i.e., no amplification), since subsequent mutations might lead to inputs with a $BAF_{L2} > 1$ for the same path. By keeping a separate maximum amplification ratio *per path*, we can find high-*BAF* requests for different vectors of the same target independently.

Amplification Maximization The amplification factor can be increased in two ways: modifying the request to lead to larger responses or finding shorter requests that result in the same response. In AMPFUZZ, we implement both strategies.

First, we leverage Angora's dataflow analysis to identify which input bytes influence the length of the response packets—concretely the `length` argument of a sending function—and then prioritize generating new inputs that mutate those bytes in particular.

Second, we add a simple yet effective mutation operator: Once the fuzzer finds a valid request, 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 or implicitly pad network packets with NUL-bytes.

3.6 Implementation

We implement AMPFUZZ on top of ParmeSan [89] and Angora [20]. As we implemented the analyzer and instrumentor components as LLVM passes, we use wllvm [3] 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 3.4.

Furthermore, as AMPFUZZ aims at amplification discovery in widespread daemon services, we add support for three essential 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 load CFGs for libraries dynamically. 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`.

4 Evaluation

We evaluate AmpFuzz in various settings to show its efficacy in finding amplification attack vectors in general and the benefits of our approaches to UDP awareness and amplification maximization. In particular, we seek to answer the following research questions:

4.1 Research Questions

RQ1 Can AmpFuzz successfully find amplification attack vectors with no a priori information about the target service? To this end, we run AmpFuzz on a set of targets, including known-vulnerable services.

RQ2 Does UDP-aware fuzzing help to find vulnerabilities faster than with the de-facto standard use of static timeouts? Here we compare the time it takes AmpFuzz to find the first amplification vector between a UDP-aware configuration and multiple static timeout values.

RQ3 Do our approaches to amplification maximization yield higher amplification factors than purely coverage guided network fuzzing? For this, we compare the maximum BAF_{L2} achieved by AmpFuzz with and without amplification maximization enabled.

4.2 Fuzz Target Selection and On-Boarding

To find suitable targets, namely UDP-based network services, we leveraged the SELinux reference policy [2]. We search for programs requesting permission to open and handle UDP ports (SELinux labels `corenet_udp_*`), which we then cross-referenced with the Debian package database [1]. Filtering for packages that we could successfully rebuild using Clang—a technical requirement for our LLVM-based instrumentation—left us with 71 candidate services over 61 packages. From these, we selected 20 services (18 packages), shown in Table 3, Appendix A, including several with previously reported amplification vulnerabilities (e.g., NTP `version`, memcached `stats`, and (x)inetd `CharGen`). While we were unable to successfully instrument the popular DNS implementations `bind` and `dnsmasq`, we included `knotd` and `stubby` instead.

For each target, we manually determined the minimal required command-line arguments and configuration files. We verified that they opened a listening UDP socket and can run inside an unprivileged Docker container. Some programs provide different services on multiple ports. Our final selection comprises 28 targets.

4.3 Experimental Setup

We fuzzed each target inside a target-specific docker image built on top of the official `debian:bullseye` base image[2]. For each run, the fuzzer was provided only the single byte seed "`a`". For UDP-aware runs, the fallback timeout was set to 500ms for both begin and end of request processing. To reduce noise in the results through randomness [43], fuzzing campaigns were repeated 5 times per target and setup. We performed all experiments on a server with 2 Intel® Xeon® Gold 6230N processors and 512 GB of RAM.

4.4 (RQ1) Efficacy of Fuzzing for Amplification Vulnerabilities

To assess the efficacy of finding amplification vectors through fuzzing, we ran AmpFuzz on each target for 24 hours. The results are shown in Table 2. For each target for which at least one request-response pair could be found, we report maximum Ethernet bandwidth amplification factor ($\max(BAF_{L2})$) and the "naive" UDP payload amplification factor ($\max(BAF_{L7})$) for comparison, *best* lists the maximum value over all 5 runs, while *mean* and *std* denote the mean and standard deviation.

Overall, we find that AmpFuzz is able to discover *true* amplification ($BAF_{L2} > 1$) in 13 and reflection vulnerabilities ($BAF_{L2} = 1$) in 6 of the 28 tested targets.

Protocols with known amplification vectors Our dataset contains 12 targets implementing protocols for which amplification vulnerabilities have previously been found through manual analysis, namely CharGen (19), DNS (53), TFTP (69), SSDP (1900), and memcached (11211). On these, AmpFuzz manages to find 7 amplifications and 3 reflections. We investigated the cases in which AmpFuzz fails to find expected amplifications: For `chronyd`, no NTP-based amplification can be found as chrony deliberately lacks support for NTP mode 6 control messages and mode 7 extensions. Likewise, `minissdpd` does not respond to `M-SEARCH` requests in the given configuration and is hence invulnerable to the known SSDP amplification vector. The CharGen implementation of openbsd-inetd `inetd` cannot be triggered in our evaluation setup, as openbsd-inetd ignores requests to the loopback interface. After manually removing this check from the openbsd-inetd source code AmpFuzz succeeds in finding the expected amplification. Lastly, the known DNS amplification vector

[2] Digest `6f4986d78878`

Table 2: 24h Fuzzing Campaign Results, each experiment was repeated 5 times. Novel vulnerabilities are highlighted.

target	port	# paths		# requests		# amps		max(BAF_{L2})		max(BAF_{L7})	
		best	mean ±std	best	mean ±std	best	mean ±std	best	mean ±std	best	mean ±std
(atftpd) atftpd	69	11582	7819.0 ±3479.7	91	70.6 ±15.1	47	38.0 ±7.2	**3.63**	3.27 ±0.44	10.00	7.68 ±2.14
(atftpd) in.tftpd	69	50	46.8 ±2.8	6	6.0 ±0.0	5	5.0 ±0.0	**1.14**	1.14 ±0.0	27.00	27.0 ±0.0
(chrony) chronyd	123	106	71.8 ±25.0	8	6.2 ±1.8	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
	323	243	234.0 ±7.7	6	6.0 ±0.0	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
(knot) knotd	53	201	152.4 ±31.6	99	57.8 ±28.8	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
(krb5-admin-server) kadmind	464	469	451.0 ±12.2	92	78.0 ±10.5	92	78.0 ±10.5	**2.91**	2.86 ±0.09	8.75	8.02 ±1.13
(memcached) memcached	11211	370	326.2 ±44.1	41	30.0 ±8.0	33	14.0 ±14.1	**32.45**	14.82 ±16.13	129.07	52.96 ±69.48
(ntp) ntpd	123	1324	1039.2 ±289.5	329	234.0 ±67.5	20	14.2 ±3.8	**7.47**	7.47 ±0.0	36.00	36.0 ±0.0
(ntpsec) ntpd	123	1427	809.8 ±455.6	244	181.8 ±59.4	10	6.0 ±2.8	**7.28**	7.28 ±0.0	35.00	32.81 ±4.38
(openafs-fileserver) bosserver	7007	1054	853.6 ±209.5	259	219.6 ±57.0	212	152.4 ±55.9	**4.59**	4.59 ±0.0	9.75	9.75 ±0.0
(stubby) stubby	53	2	2.0 ±0.0	1	1.0 ±0.0	-		1.00	1.0 ±0.0	∞	
(talkd) in.ntalkd	518	44	41.4 ±2.2	22	20.0 ±1.9	1	1.0 ±0.0	**1.09**	1.09 ±0.0	24.00	24.0 ±0.0
(talkd) in.talkd	517	44	41.6 ±2.6	22	19.6 ±1.8	1	1.0 ±0.0	**1.09**	1.09 ±0.0	24.00	24.0 ±0.0
(tftpd) in.tftpd	69	1297	980.2 ±272.9	28	23.6 ±2.7	22	20.2 ±1.3	**1.14**	1.14 ±0.0	13.50	13.5 ±0.0
(xinetd) xinetd	7	3	3.0 ±0.0	1	1.0 ±0.0	-		1.00	1.0 ±0.0	1.00	1.0 ±0.0
	13	6	6.0 ±0.0	1	1.0 ±0.0	1	1.0 ±0.0	**1.12**	1.12 ±0.0	∞	
	19	3	3.0 ±0.0	1	1.0 ±0.0	1	1.0 ±0.0	**16.72**	16.72 ±0.0	∞	
	37	3	3.0 ±0.0	1	1.0 ±0.0	-		1.00	1.0 ±0.0	∞	
(xl2tpd) xl2tpd	1701	164	83.2 ±59.2	75	35.2 ±30.2	10	2.4 ±4.3	**3.52**	2.19 ±1.15	5.81	3.21 ±2.49

(implementations knotd and stubby) utilizes ANY requests for a domain with large DNS records. As such, it not only requires the resolver (i.e., the target) to be able to perform upstream queries (which we had disabled in our test setup), but also knowledge of a valid domain with active records.

Novel amplification vectors AmpFuzz also discovers 9 previously unknown reflection and amplification vulnerabilities. Next to the trivial reflections in the legacy echo (7), daytime (13), time (37) and talk (517)/ntalk(518) protocols, these also include non-trivial reflections in control protocol of chronyd (323) and amplifications in the Kerberos administration server kadmind, the OpenAFS Basic OverSeer server bosserver (see Section 4.7), and in the layer 2 tunneling protocol [83] implementation xl2tpd.

Estimating individual vulnerabilities To quantify the number of distinct amplification vectors per target, we not only report the number of unique program traces (*# paths*). We also count the number of distinct request handling behaviors, both overall (*# requests*) and for requests leading to amplification (*# amps*). For this, we leverage data collected by the dataflow framework during fuzzing, from which we extract the set of all request-dependent CFG edges for each input. That way, two requests that exercise different parts of the target are counted individually, while requests whose traces differ only in the iteration count of a loop or similar are only counted as one. For example, atftpd uses a nested loop structure to tokenize and parse request options, resulting in a large number of possible execution paths. Yet, as further request handling only depends on *which* options were specified, the number of distinct request types is much lower.

Interestingly, many targets amplify traffic for multiple request types, i.e., offer multiple amplification vectors. This further highlights the need for systematic discovery of amplification vectors as simple defenses blocking only fixed request patterns may be incomplete.

We manually investigated cases where only a small number of paths were discovered during fuzzing: As we limit outgoing connections, stubby, a DNS stub resolver, always replies with SERVFAIL, irregardless of the query. For xinetd, we find that the built-in services have very low complexity, with only one or two branches during request handling. On these, AmpFuzz thus quickly achieves full path-coverage.

4.5 (RQ2) Impact of UDP-aware fuzzing

To evaluate whether UDP-aware fuzzing leads to faster findings than using de-facto standard static timeouts, we performed additional experiments running AmpFuzz on all fuzz

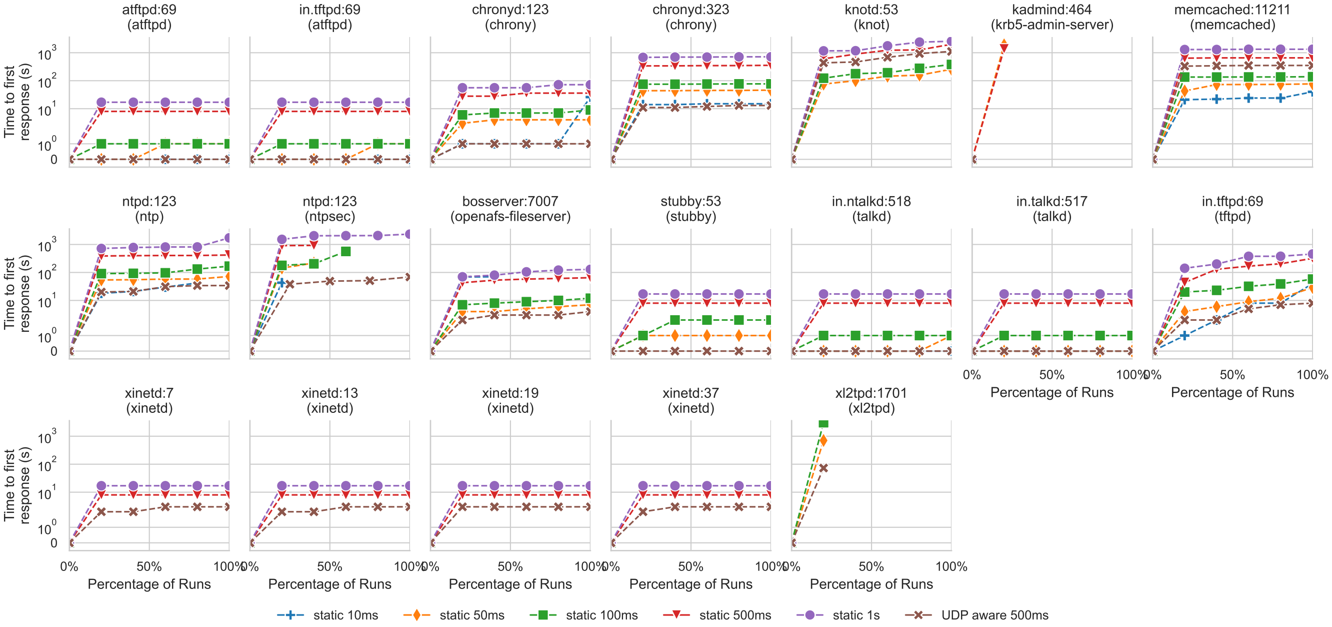


Figure 4: Time to first response for UDP-aware fuzzing vs. static timeouts (lower is better)

targets for one hour in six different configurations: UDP aware (with a default timeout of 500ms) and static timeouts of 10, 50, 100, 500, and 1000ms.

Figure 4 shows the time until our fuzzer found the first request-response pair over the experiment repetitions. As already hypothesized in Section 3.4, the optimal static timeout varies between services. For example, on `knotd` a static timeout of 50ms shows the best performance while the same timeout fails to find any responses on `xinetd`. On the other hand, while the fuzzer found request-response pair successfully with a large timeout of 1000ms, it slows down fuzzing by multiple orders of magnitude. **In contrast, in almost all cases, UDP-aware fuzzing performs as well as or better than the best performing static timeout.**

We manually analyzed the two exceptions to this, `knot` and `memcached`. In both cases, our approach to detect the end of request processing fails to entirely terminate the target due to other active background threads (e.g., for garbage collection in the case of `memcached`).

4.6 (RQ3) Amplification Maximization

Lastly, we performed additional experiments to measure the impact of our amplification maximization efforts as described in Section 3.5. For this, we ran AMPFUZZ on all fuzz targets in two configurations, once with and once without amplification maximization, and measured the maximum Ethernet amplification factor that was found after one hour, the results of which are shown in Figure 5.

Surprisingly, it appears that purely coverage-based guidance is already sufficient for some targets to find maximal amplification requests, since new request *types* also lead to new coverage. Further, by foregoing amplification maximization, more time can be spent on exploring new coverage. Yet, in almost all cases, amplification maximization allowed AMPFUZZ to find an equally large or larger maximum amplification factor after one hour. This holds true in particular for `memcached` and `bosserver`, where the *BAF* can be increased by reducing the request size *without* providing new coverage. **Thus, allocating some time-budget to amplification-maximizing queries provides a net benefit overall.**

4.7 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 1).

Packets of type `RX_PACKET_TYPE_DEBUG` and with the `RX_CLIENT_INITIATED` flag set can be used to query for debugging 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

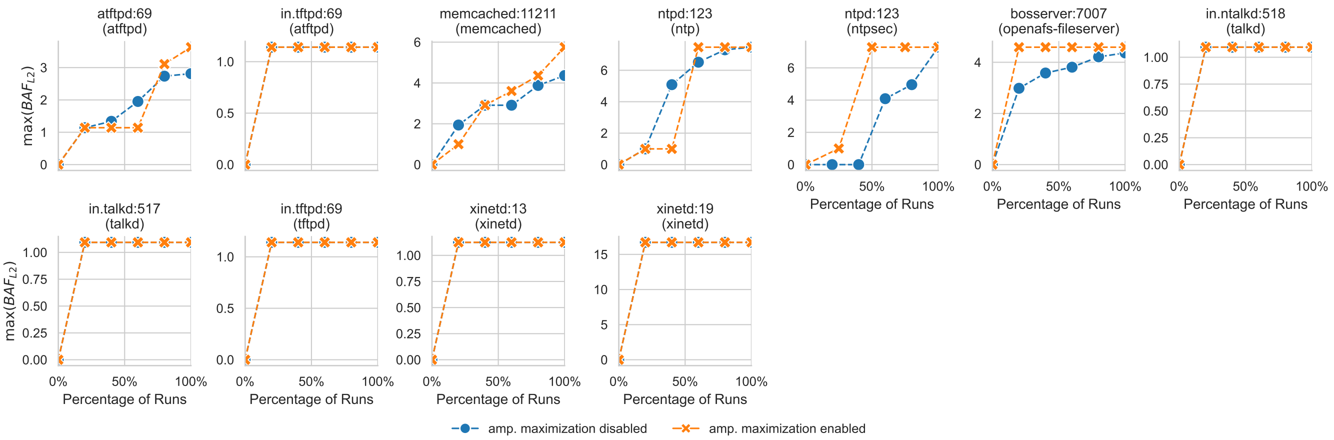


Figure 5: Cumulative distribution of maximum amplification factors over all test runs (higher is better)

```
struct rx_header {  // HEADER
        afs_uint32 epoch;
        afs_uint32 cid;
        afs_uint32 callNumber;
        afs_uint32 seq;
        afs_uint32 serial;
        u_char type;
        u_char flags;
        u_char userStatus;
        u_char securityIndex;
        u_short serviceId;
        u_short spare;
};
struct rx_debugIn { // DEBUG PAYLOAD
        afs_int32 type;
        afs_int32 index;
};
```

Listing 1: Packet structure used by OpenAFS bosserver

(`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 [74].

After contacting the maintainers of OpenAFS about our newly found amplification vector they promptly confirmed our findings. Interestingly, they informed us that this particular amplification vulnerability found by AMPFUZZ not only affects the BOS Server but all OpenAFS services sharing the same underlying RX RPC mechanism [88].

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

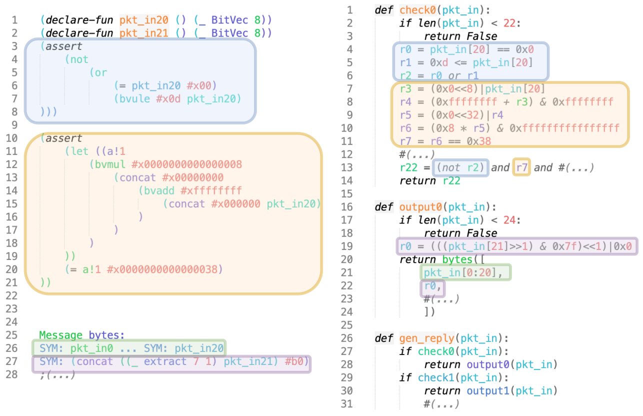
4.8 Case Study: Honeypot Synthesis

On the defensive side, amplification honeypots have proven as an invaluable tool. By mimicking the behavior of vulnerable systems, they hope that attackers discover and abuse them as reflectors. As such, they not only allow monitoring and studying attacks in real-time [44, 82], but also form the basis of several traceback mechanisms [31, 46–48].

However, creating such honeypot systems demands substantial manual effort. Since running 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 which requests the honeypot should respond to and how to compute the response. In this section, we thus present an automated honeypot synthesis approach based on AMPFUZZ.

4.8.1 Honeypot Synthesis Overview

In essence, a honeypot has to *check* incoming requests against a set of known patterns, and, if a match is found, *output* a corresponding response. The honeypot synthesis problem can thus be reduced to providing indicator `check` functions and corresponding `output` functions. Since AMPFUZZ provides us with a list of request-response pairs, the core idea of our honeypot synthesis is to use symbolic execution to capture path constraints on the request and an abstract symbolic expression of the generated response, from which we can generate these functions.



```
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 6: Honeypot code example for OpenAFS `bosserver`

Symbolic Execution To collect path constraints and output expressions, we leverage the state-of-the-art LLVM symbolic execution framework SymCC [71], which can be nicely integrated with AMPFUZZ, as both rely on LLVM IR for instrumentation. We extend SymCC by providing symbolic wrappers for receiving and sending network functions. We generate new symbolic bytes for every byte read from the correct UDP socket for receiving functions. This allows SymCC to treat network requests as symbolic inputs. For sending functions, we record all collected path constraints and the symbolic expression for every response byte. At this point, the path constraints capture precisely which conditions the request has to fulfill 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.

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. 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 model expressions 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 and additional code to ensure the correct bitwidth, while a cache ensures that equivalent subtrees are converted only once.

Figure 6 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. 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. Our generated honeypot code translates this 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.

4.8.2 Synthesized Honeypot Evaluation

As a small-scale evaluation, we synthesized a honeypot for all vulnerabilities and reflections discovered by AMPFUZZ. To ensure that our synthesized honeypot works as intended, we compared its responses to those of the original services. In all cases, the honeypot generated a response when the original service did. While responses between the two were indistinguishable in many cases on a byte-level, we also observed variance in others. This is expected for services that include, e.g., random session identifiers in their responses 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. Still, automatically synthesized honeypots can be deployed quickly to monitor the exploitation of new amplification vulnerabilities.

4.9 Comparison with AmpMap [59]

In a recent study, Moon et al. [59] show how to estimate the global "amplification risk" posed by amplification vulnerabilities. For this, they develop AmpMap, a tool that probes public Internet servers for amplification vulnerabilities in 6 UDP-based protocols. As AmpMap generates requests for these protocols based on protocol descriptions, it can be seen as an instance of grammar-based blackbox fuzzing. We, therefore, compare its approach and findings to that of AMPFUZZ.

Instrumentation and Configuration In contrast to AMPFUZZ, AmpMap does not require to instrument target services, which in turn enables probing real-world systems that may have multiple configurations.

Grammar-based Fuzzing By deriving inputs from a formal specification, grammar-based fuzzing promises to generate *valid* inputs only, thus allowing a fuzzer to spend more time testing meaningful inputs rather than fighting input syntax. While many networking protocols are specified (at least in a human-readable form) in RFCs, this is not always true. One example is the control message protocol used by chrony (323), which is only specified in the source code of chrony itself. While AMPFUZZ found a reflection vulnerability for this protocol, AmpMap cannot generate any requests for this target without a protocol specification. In other cases, protocols allow for custom extensions. For example, NTP `monlist` [6] is a private mode 7 extension by (ntp) `ntpd`. The only reason AmpMap can still identify servers vulnerable to `monlist` is that it leverages scapy [13] for request generation, which includes a model of these extensions based on a review of the ntpd source code.

Interestingly, in some cases, AmpMap *fails* to find amplification vulnerabilities even when a protocol specification is available. This is the case for NTP `read variables`, the most severe NTP mode 6 vulnerability. Specifically, mode 6 control messages consist of a fixed-size header and a variable length data field. For the `read variables` command, (ntp) `ntpd` requires that the 16-bit header fields `offset` and `assocID` fields are set to 0, that the 16-bit header field `count` corresponds to the length of the data field, and that the request is padded to a multiple of 4 bytes. However, AmpMap's grammar does not take into account the specified link between `count` and the data field, and further always adds a fixed data field of 5 bytes, which violates the padding constraint. Yet, even with a fixed empty data and a constant `count` of 0, the random black-box approach of AmpMap only generates valid `read variables` requests with a chance of $1 : 2^{32}$.

Lastly, there are instances where a request's amplification factor can be increased by *violating* the protocol description. An instance of this is the vulnerability detailed in Section 4.7. Here, AMPFUZZ was able to omit the last four bytes from the request, although the Rx protocol draft [88] considers all parts of the debug request payload as non-optional.

Overall, we thus find that the lack of a protocol specification excludes fuzz targets, that incomplete protocol specifications miss vulnerabilities, and that vulnerabilities exist even outside of complete protocol specifications.

Expert Knowledge AmpMap further augments the underlying, generic protocol specifications with expert knowledge of concrete vulnerabilities. For example, while the DNS specification only defines how domain names need to be encoded, AmpMap restricts the choice to 10 active domain names for which DNS records exist. This ensures that all generated DNS requests will generate responses. Likewise, for SSDP, the grammar used by AmpMap is restricted to `M-SEARCH` requests only, for which a known vulnerability exists. Yet, providing such expert knowledge for untested protocols is a laborious task akin to analyzing the protocol by hand. In addition, any such restrictions will limit the scope of amplifications to small parts of the vulnerable programs only.

5 Discussion

In this section, we outline shortcomings of our evaluation and how they can be tackled, discuss the underlying assumptions made by AMPFUZZ, and describe how we adhere to the best ethical standards during our active measurements and by disclosing the vulnerabilities to vendors.

5.1 Evaluation Shortcomings

LLVM IR 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 evaluation, we further relied on wllvm [3], thus restricting our dataset to services written in C/C++. Fortunately, this includes most network daemons on Linux. However, we noted a few special cases where services were implemented in scripting languages such as Perl or used gcc-specific extensions such as inline assembly.

UDP Sockets Another limitation stems from our choice of using "real" UDP sockets for passing in- and output between the fuzzer and the SuT. This ensures that all socket-related APIs, especially those relying on socket states such as `poll` and `select`, behave as they would in real-world scenarios. However, 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 some targets that, e.g., attempted to perform additional socket configuration using `ioctl` calls. Actual sockets also impact parallelization during fuzzing, as only one socket may be bound to the same port and address at a time.

We could potentially avoid both problems by preventing the SuT from binding "real" sockets and hooking the relevant socket API functions instead, albeit at the cost of 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.

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.

5.2 Limitations

Single UDP Request Model Not all amplification vectors can be discovered using AmpFuzz. In particular, AmpFuzz assumes that requests are sent via UDP and that a single request suffices to trigger amplification. While these assumptions currently hold for the vast majority of known vulnerabilities, TCP-based amplification is possible [15, 51] and attackers have reportedly used preparatory TCP requests to implant large payloads on memcached amplifiers [23]. As such, non-UDP and multi-request amplifications are currently out-of-scope for AmpFuzz.

Request Complexity Other amplification vectors are not well-suited for discovery through greybox fuzzing. This includes, for example, DNS, where valid domain names can hardly be found without expert knowledge, but also LDAP [77], where requests must be ASN.1 BER encoded[3]. Yet, where a formal protocol specification exists, grammar-based fuzzing approaches [59] can still handle the latter case.

Instrumentation AmpFuzz assumes that targets can be readily instrumented and is hence unable to fuzz closed-source programs. However, recent advances in binary-only fuzzing [61] might enable searching for amplification vectors in closed-source programs in the future.

Target Configuration As noted in Section 4.2, AmpFuzz requires some manual onboarding for each target to determine the target's command line arguments and configuration options. While in many cases, we can complete this process with a cursory look at the services' man page in only a few minutes, it remains a manual process. Furthermore, the amplification potential of a target can differ per configuration [59]. Therefore, AmpFuzz would be best suited for large-scale deployment in an approach similar to OSS-Fuzz [39], which invites software maintainers to provide their fuzzing configurations and automates everything from there.

5.3 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 [28] 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. Our institute's ERB approved all our active experiments. 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.

5.4 Coordinated Disclosure

Where possible, we contacted the maintainers of affected packages before submitting this paper to disclose our findings. This ensures that they have a minimum of 90 days before our findings are publicly disclosed, which aligns with the industry standard. No party asked us to redact our results before submission.

6 Related Work

Amplification DDoS and (network) fuzzing have been active fields of research in the past. We now discuss how previous works from these areas relate to AmpFuzz.

6.1 Amplification DDoS

Paxson first discovered the risk of abusing third-party services as reflectors for DDoS attacks in 2001 [66], 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 [74].

Following that, several 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 [84], while MacFarland et al. analyzed how an attacker can optimize their queries to achieve larger amplification factors [56]. Liu et al. and Adamsky et al. both show how peer-to-peer networks can be leveraged to launch amplification attacks [9,55], including a scenario in which the attacker first uploads data to a distributed storage system and later spoofs download requests from the victim. This attack is conceptually similar to the later discovered Memcached amplification attack [23]. Beyond UDP reflection, Kührer et al. investigate the amplification potential of the TCP handshake itself [51], Sargent et al. that of the IGMP management protocol [76], while Gasser et al. warn about the threat posed by publicly reachable BACnet devices [35].

Yet, all of the amplification vectors found in the works above were found exclusively through manual protocol specification review or reverse engineering. Our goal is to automatize new amplification discovery.

Knowing amplification vulnerabilities is vital for several reasons. For one, it allows for assessing the threat landscape by scanning for potential amplifiers or monitoring (malicious) scanning activities using network telescopes [25,49]. Furthermore, once a vulnerability is known, steps can be taken to mitigate it. For instance, Kührer et al. [50] report a reduction

[3] X.690, https://www.itu.int/rec/T-REC-X.690/

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. [53] also measures the remediation effects on other protocols. Lastly, knowledge of amplification vulnerabilities also enables passive monitoring of attacks through DDoS honeypots. As such, amplification DDoS honeypots have been proposed by Krämer et al. in 2015 [44] and by Thomas et al. in 2017 [82].

Honeypots have also been used to provide an additional perspective on the Denial-of-Service ecosystem, e.g., by Jonker et al. [41], while another line of work attempts to provide honeypot-based traceback capabilities [31, 46, 47]. By providing law enforcement agencies with additional leads when investigating attacks, the latter underlines the utility of honeypot systems also outside the research community.

6.2 Algorithmic Complexity DoS

Fuzzing has been successfully applied to find another class of Denial-of-Service attacks, namely through Algorithmic Complexity (AC) bugs that incur expensive resource usage when processing certain inputs. In 2017, Petsios et al. presented SlowFuzz [69] based on libFuzzer [7] that discovers such inputs for C programs using evolutionary fuzzing: the number of executed basic blocks is counted for each randomly generated input, and the top ones are staged for one mutation each. The same year, Lemieux et al. [52] improved over SlowFuzz's results, presenting a fuzzing technique based on AFL [87] which applies several mutation transformations to random inputs prioritizing those traversing the most CFG edges. Noller et al. [63] further demonstrate how fuzzing can be supplemented with symbolic execution to uncover deep execution paths with high computational resource consumption. In 2020, Blair et al. [14] challenged the coverage of AFL-based approaches. They presented micro-fuzzing for Java programs which allows for identifying the AC-vulnerability triggering inputs for individual functions instead of the whole program, uncovering previously unknown AC bugs.

However, AC bugs and amplification vulnerabilities form two different attack classes: While AC bugs can be used to cause logical denial-of-service attacks of the vulnerable system, amplification attacks abuse vulnerable systems as intermediaries to attack *other* systems.

6.3 Fuzzing Employing Symbolic Execution

While we only use symbolic execution to generate honeypots, several approaches have demonstrated that symbolic execution can also assist fuzzing. Such hybrid fuzzing approaches, popularized by Stephens et al. [80], can potentially reach "deeper" code paths into the SuT by using constraint solvers to find new inputs. Yet, as symbolic execution is expensive, several approaches aim to use it sparingly. For example, Peng et. al. [67] perform regular fuzzing on a simplified version of the SuT that lacks some checks and only identify with symbolic execution the feasibility of discovered paths in the original program, while Liang et al. [54] recently proposed to use it for the initial seed generation only.

6.4 Network Fuzzing

With the recent trend of software fuzzing, some fuzzers have been developed to target network daemons. Next to general-purpose fuzzers that simply use network sockets as other means of providing input to the SuT [10, 27, 37, 40, 64], this includes dedicated network fuzzers which generate inputs either based on previously recorded client-server interactions [34, 75, 81] or from protocol descriptions [11, 30, 42, 68, 79]. Some further attempt to infer server-side state in order to reach code paths that require multiple messages between a client and the server [29, 34, 70]. Most of these only target TCP services, where terminated connections can be observed easily. The ones which allow for fuzzing UDP services either ignore replies from the server completely [10, 40], or rely on timeouts [30, 37, 64, 68, 70, 75, 81] or user-provided target-specific scripts [27]. However, as shown in Section 4.4, UDP-awareness of AMPFUZZ outperformed simple timeout-based solutions by multiple orders of magnitude. More importantly, these previous fuzzers aim to find either inputs that lead to server-side crashes or 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 Conclusion

AMPFUZZ is the first protocol-agnostic approach to discover amplification DDoS vulnerabilities in UDP-based network services systematically. 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 significantly improves fuzzing performance. After finding candidate daemons through an SELinux reference policy analysis, we evaluated 28 daemons extracted from the Debian package repositories. In total, AMPFUZZ identified vulnerabilities in 19 network services, 13 out of which provide amplification with $BAF_{L2} > 1$. Next to rediscovering 7 known vulnerabilities, our principled approach revealed 9 previously unknown vulnerabilities. For the most severe of these, with a non-trivial 4.59 BAF_{L2}, we further show its real-world amplification potential through an Internet-wide scan.

Acknowledgments

The authors sincerely thank the anonymous reviewers and their shepherd, Sang Kil Cha, for their valuable feedback and suggestions which helped to improve the paper.

References

[1] Debian packages repo. https://www.debian.org/distrib/packages.

[2] SELinux project. https://github.com/SELinuxProject.

[3] Whole program LLVM. https://github.com/SRI-CSL/whole-program-llvm. Version 1.2.8.

[4] Routing Information Protocol. Technical Report 1058, June 1988.

[5] Network Time Protocol (Version 3) Specification, Implementation and Analysis. Technical Report 1305, March 1992.

[6] CVE-2013-5211. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-5211, 2013.

[7] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2022.

[8] Joe Abley, Ólafur Guðmundsson, Marek Majkowski, and Evan Hunt. Providing Minimal-Sized Responses to DNS Queries That Have QTYPE=ANY. Technical Report 8482, January 2019.

[9] Florian Adamsky, Syed Ali Khayam, Rudolf Jäger, and Muttukrishnan Rajarajan. 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.

[10] Dave Aitel. SPIKE, a fuzzer creation kit. http://www.immunitysec.com/downloads/SPIKE2.9.tgz. Last Accessed 2021-06-01.

[11] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin C. Almeroth, Richard A. Kemmerer, and Giovanni Vigna. SNOOZE: toward a stateful network protocol fuzzer. In *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*. Springer, 2006.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[13] Philippe Biondi. Scapy - packet crafting for python2 and python3. https://scapy.net/. Last Accessed 2022-02-01.

[14] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[15] Kevin Bock, Abdulrahman Alaraj, Yair Fax, Kyle Hurley, Eric Wustrow, and Dave Levin. Weaponizing middleboxes for TCP reflected amplification. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, 2021.

[16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 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*. ACM, 2016.

[17] Bill Brenner. RIPv1 reflection DDoS making a comeback. https://blogs.akamai.com/2015/07/ripv1-reflection-ddos-making-a-comeback.html, 2015.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 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*. USENIX Association, 2008.

[19] Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. 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*. ACM, 2018.

[20] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, May 2018. IEEE.

[21] Maria Christakis, Peter Müller, and Valentin Wüstholz. 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*. ACM, 2016.

[22] Catalin Cimpanu. AWS said it mitigated a 2.3 Tbps DDoS attack, the largest ever. https://www.zdne

t.com/article/aws-said-it-mitigated-a-2-3-tbps-ddos-attack-the-largest-ever/. Last Accessed 2022-02-01.

[23] Cloudflare. Memcached DDoS attack. https://www.cloudflare.com/en-gb/learning/ddos/memcached-ddos-attack/.

[24] Cloudflare. Rfc8482 - saying goodbye to ANY. https://blog.cloudflare.com/rfc8482-saying-goodbye-to-any/, 2019.

[25] Jakub Czyz, Michael Kallitsis, Manaf Gharaibeh, Christos Papadopoulos, Michael Bailey, and Manish Karir. 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*, 2014.

[26] Amir Dahan. Business as usual for azure customers despite 2.4 Tbps DDoS attack. https://azure.microsoft.com/en-us/blog/business-as-usual-for-azure-customers-despite-24-tbps-ddos-attack/. Last Accessed 2022-02-01.

[27] denandz. Fuzzotron - a fuzzing harness built around ouspg's blab and radamsa. https://github.com/denandz/fuzzotron. Last Accessed 2021-06-01.

[28] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 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.

[29] Rong Fan and Yaoyao Chang. 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.

[30] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. Analysis of DTLS implementations using protocol state fuzzing. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, 2020.

[31] Osvaldo L. H. M. Fonseca, Ítalo Cunha, Elverton C. Fazzion, Wagner Meira Jr., Brivaldo Junior, Ronaldo A. Ferreira, and Ethan Katz-Bassett. Tracking down sources of spoofed IP packets. In *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, 2020.

[32] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018.

[33] Vijay Ganesh, Tim Leek, and Martin C. Rinard. Taint-based directed whitebox fuzzing. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009.

[34] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 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.

[35] Oliver Gasser, Quirin Scheitle, Benedikt Rudolph, Carl Denis, Nadja Schricker, and Georg Carle. The amplification threat posed by publiclyreachable bacnet devices. *J. Cyber Secur. Mobil.*, 6(1), 2017.

[36] Xi Ge, Kunal Taneja, Tao Xie, and Nikolai Tillmann. 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*. ACM, 2011.

[37] GitLab. GitLab protocol fuzzer. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce. Last Accessed 2021-06-01.

[38] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3), 2012.

[39] Google. Oss-fuzz. https://google.github.io/oss-fuzz/. Last Accessed 2022-02-01.

[40] Dean Jerkovich. rage against the network. https://github.com/deanjerkovich/rage_fuzzer. Last Accessed 2021-06-01.

[41] Mattijs Jonker, Alistair King, Johannes Krupp, Christian Rossow, Anna Sperotto, and Alberto Dainotti. 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.

[42] Rauli Kaksonen, Marko Laakso, and Ari Takanen. System security assessment through specification mutations and fault injection. In *Communications and Multimedia Security Issues of the New Century, Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues, May 21-22,*

2001, Darmstadt, Germany, volume 192 of *IFIP Conference Proceedings*. Kluwer, 2001.

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018.

[44] Lukas Krämer, Johannes Krupp, Daisuke Makita, Tomomi Nishizoe, Takashi Koide, Katsunari Yoshioka, and Christian Rossow. 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*, volume 9404 of *Lecture Notes in Computer Science*. Springer, 2015.

[45] Brian Krebs. DDoS on dyn impacts twitter, spotify, reddit. https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/, 2016.

[46] Johannes Krupp, Michael Backes, and Christian Rossow. 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.

[47] Johannes Krupp, Mohammad Karami, Christian Rossow, Damon McCoy, and Michael Backes. 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*, volume 10453 of *Lecture Notes in Computer Science*. Springer, 2017.

[48] Johannes Krupp and Christian Rossow. Bgpeek-a-boo: Active bgp-based traceback for amplification ddos attacks. In *6th IEEE European Symposium on Security and Privacy*, September 2021.

[49] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. 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.

[50] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. 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*. USENIX Association, 2014.

[51] Marc Kührer, Thomas Hupperich, Christian Rossow, and Thorsten Holz. 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.

[52] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Amsterdam Netherlands, July 2018. ACM.

[53] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, 2016.

[54] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. Deepfuzzer: Accelerated deep greybox fuzzing. *IEEE Trans. Dependable Secur. Comput.*, 18(6), 2021.

[55] Bingshuang Liu, Jun Li, Tao Wei, Skyler Berg, Jiayi Ye, Chen Li, Chao Zhang, Jianyu Zhang, and Xinhui Han. Sf-drdos: The store-and-flood distributed reflective denial of service attack. *Comput. Commun.*, 69, 2015.

[56] Douglas C. MacFarland, Craig A. Shue, and Andrew J. Kalafut. The best bang for the byte: Characterizing the potential of DNS amplification attacks. *Comput. Networks*, 116, 2017.

[57] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Software Eng.*, 47(11), 2021.

[58] Phil McMinn. Search-based software test data generation: a survey. *Softw. Test. Verification Reliab.*, 14(2), 2004.

[59] Soo-Jin Moon, Yucheng Yin, Rahul Anand Sharma, Yifei Yuan, Jonathan M. Spring, and Vyas Sekar. Accurately measuring global risk of amplification attacks using ampmap. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, 2021.

[60] Carlos Morales. NETSCOUT arbor confirms 1.7 Tbps DDoS attack; the terabit attack era is upon us. https://www.netscout.com/blog/asert/netscout-arbor-confirms-17-tbps-ddos-attack-terabit-attack-era. Last Accessed 2022-02-01.

[61] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security*

Symposium, USENIX Security 2021, August 11-13, 2021, 2021.

[62] Netscout. Netscout threat intelligence report - issue 7: Findings from 1h 2021. https://www.netscout.com/sites/default/files/2021-09/ThreatReport_1H2021_FINAL.pdf, 2021. Last Accessed 2022-02-01.

[63] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. Badger: Complexity analysis with fuzzing and symbolic execution. In *Software Engineering and Software Management, SE/SWM 2019, Stuttgart, Germany, February 18-22, 2019*, 2019.

[64] Open Reverse Code Engineering. Sulley: A pure-python fully automated and unattended fuzzing framework. https://github.com/OpenRCE/sulley. Last Accessed 2021-06-01.

[65] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: domain-specific fuzzing with waypoints. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), October 2019.

[66] Vern Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *Comput. Commun. Rev.*, 31(3), 2001.

[67] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018.

[68] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz. Last Accessed 2021-06-01.

[69] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA, October 2017. ACM.

[70] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 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.

[71] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020.

[72] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 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*. The Internet Society, 2017.

[73] Jonathan Respeto. New DDoS vector observed in the wild: Wsd attacks hitting 35/Gbps. https://www.akamai.com/blog/security/new-ddos-vector-observed-in-the-wild-wsd-attacks-hitting-35gbps, 2019.

[74] Christian Rossow. 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*. The Internet Society, 2014.

[75] Dobin Rutishauser. Fuzzing for worms. https://github.com/dobin/ffw. Last Accessed 2021-06-01.

[76] Matthew Sargent, John Kristoff, Vern Paxson, and Mark Allman. On the potential abuse of IGMP. *Comput. Commun. Rev.*, 47(1), 2017.

[77] Jim Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. Technical Report 4511, June 2006.

[78] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin C. Rinard. 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*. ACM, 2015.

[79] Congxi Song, Bo Yu, Xu Zhou, and Qiang Yang. Sp-fuzz: A hierarchical scheduling framework for stateful network protocol fuzzing. *IEEE Access*, 7, 2019.

[80] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA, 2016. Internet Society.

[81] Cisco Talos. Mutiny fuzzing framework. https://github.com/Cisco-Talos/mutiny-fuzzer. Last Accessed 2021-06-01.

[82] Daniel R. Thomas, Richard Clayton, and Alastair R. Beresford. 1000 days of UDP amplification ddos attacks. In *2017 APWG Symposium on Electronic Crime*

Research, eCrime 2017, Phoenix, AZ, USA, April 25-27, 2017, 2017.

[83] Andrew J. Valencia, Glen Zorn, William Palter, Gurdeep-Singh Pall, Mark Townsley, and Allan Rubens. Layer Two Tunneling Protocol "L2TP". Technical Report 2661, August 1999.

[84] Roland van Rijswijk-Deij, Anna Sperotto, and Aiko Pras. 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*, 2014.

[85] Paul Vixie. Response rate limiting in the domain name system (dns rrl). http://www.redbarn.org/dns/ratelimits, 2012.

[86] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010.

[87] Michal Zalewski. American Fuzzy Lop: a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/, 2010.

[88] Nickolai Zeldovich. Rx protocol specification draft. http://web.mit.edu/kolya/afs/rx/rx-spec. Last Accessed 2022-02-01.

[89] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. USENIX Association, August 2020.

A Selected Fuzz Targets

Table 3 presents all targets used in our evaluation discussed in Section 4.

Table 3: Selected Fuzz Targets (marked ports denote protocols with known amplification vectors)

package version	binary	ports
atftpd 0.7.git20120829-3.3+deb11u1	atftpd	69
	in.tftpd	69
chrony 4.0-8+deb11u1	chronyd	123, 323
inetutils-syslogd 2:2.0-1	syslogd	514
knot 3.0.5-1	knotd	53
krb5-admin-server 1.18.3-6+deb11u1	kadmind	464
krb5-kdc 1.18.3-6+deb11u1	krb5kdc	88
memcached 1.6.9+dfsg-1	memcached	11211
minissdpd 1.5.20190824-1	minissdpd	1900
ntp 1:4.2.8p15+dfsg-1	ntpd	123
ntpsec 1.2.0+dfsg1-4	ntpd	123
openafs-fileserver 1.8.6-5	bosserver	7007
openbsd-inetd 0.20160825-5	inetd	7, 13, 19, 37
rsyslog 8.2102.0-2	rsyslogd	514
stubby 1.6.0-2	stubby	53
talkd 0.17-17	in.ntalkd	518
	in.talkd	517
tftpd 0.17-23	in.tftpd	69
xinetd 1:2.3.15.3-1+b1	xinetd	7, 9, 13, 19, 37
xl2tpd 1.3.12-1.1	xl2tpd	1701