Attacking the Network Time Protocol

Aanchal Malhotra, Isaac E. Cohen, Erik Brakke, and Sharon Goldberg
{aanchal4, icohen93, ebrakke}@bu.edu, goldbe@cs.bu.edu
Boston University, Boston, MA.

***Abstract*—We explore the risk that network attackers can exploit unauthenticated Network Time Protocol (NTP) traffic to alter the time on client systems. We first discuss how an on-path attacker, that hijacks traffic to an NTP server, can quickly shift time on the server's clients. Then, we present a extremely low-rate (single packet) denial-of-service attack that an off-path attacker, located anywhere on the network, can use to disable NTP clock synchronization on a client. Next, we show how an off-path attacker can exploit IPv4 packet fragmentation to shift time on a client. We discuss the implications on these attacks on other core Internet protocols, quantify their attack surface using Internet measurements, and suggest a few simple countermeasures that can improve the security of NTP.**

I. Introduction

NTP [43] is one of the Internet's oldest protocols, designed to synchronize time between computer systems communicating over unreliable variable-latency network paths. NTP has recently received some attention from security researchers due to software-implementation flaws [50], [58], and its potential to act as an amplifier for distributed denial of service (DDoS) attacks [14], [67]. However, the community still lacks visibility into the robustness of the NTP ecosystem itself, as well as the integrity of the timing information transmitted by NTP. These issues are particularly important because time is a fundamental building block for computing applications, and is heavily utilized by many cryptographic protocols.

NTP most commonly operates in an hierarchical client-server fashion. Clients send queries to solicit timing information from a set of preconfigured servers that usually remain static over time. While NTP supports both symmetric and asymmetric cryptographic authentication [23], in practice, these modes of operation are rarely used (Section III).

Our goal is therefore to explore attacks on unauthenticated NTP that are possible *within* the NTP protocol specification [43]. We consider both (1) *on-path attacks*, where the attacker occupies a privileged position on the path between NTP client and one of its servers, or hijacks (with *e.g.,* DNS [26], [27] or BGP [15], [21], [52]) traffic to the server, and (2) *off-path attacks*, where the attacker can be anywhere on the network and does not observe the traffic between client and any of its servers. This paper considers the following:

This is the authors' full version of the paper that will appear at NDSS '16, 21-24 February 2016, San Diego, CA, USA. First disclosure: August 20, 2015. First public posting: October 21, 2015. Last update: November 2, 2015. http://eprint.iacr.org/2015/1020

Implications (Section II). We consider a few implications of attacks on NTP, highlighting protocols and applications whose correctness and security relies on the correctness of local clocks. We discuss why some applications (*e.g.,* authentication, bitcoin, caching) can fail if time is shifted by just hours or days, while others (*e.g.,* TLS certificates, DNSSEC) fail when time is shifted by months or years.

Dramatic time steps by on-path attackers (Sections IV). We discuss various techniques that an on-path attacker who intercepts traffic to an NTP server can use to shift time on its clients by hours or even years. Our attacks exploit NTP's behavior upon initialization, as well as the fact than an on-path attacker can easily determine exactly when an ntpd client is initializing. We also present "small-step-big-step" attack that stealthily shifts client clocks when clients are unlikely to notice; this behavior has been captured in CVE-2015-5300.

Off-path denial-of-service attack (Section V-C). We show how an off-path attacker can disable NTP at a victim client by exploiting NTP's rate-limiting mechanism, the *Kiss-o'-Death (KoD)* packet. Our attacker need only spoof a *single* KoD packet from each of the client's preconfigured servers. The client stops querying its servers and is unable to update its local clock. The current NTP reference implementation is vulnerable to this attack, which is described in CVE-2015-7704. An off-path attacker that uses standard networking scanning tools (*e.g.,* zmap [18]) to spoof KoD packets can launch this attack on most NTP clients in the Internet within a few hours.

Time steps by off-path attackers. Next, we consider off-path attackers that step time on victim NTP clients:

1. Pinning to bad timekeepers (Section V-D). We first consider an off-path attackers that uses spoofed KoD packets to force clients to synchronize to malfunctioning servers that provide incorrect time; we find that NTP is pretty good at preventing this type of attack, although it succeeds in certain situations.

2. Fragmentation attack (Section VI). Then we show how NTP's interaction with lower layer protocols (ICMP, IPv4) can be exploited in a new off-path IPv4 fragmentation attack that shifts time on a victim client. We explain why NTP's clock discipline algorithms require our attack to craft a *stream* of self-consistent packets (rather than just one packet, as in [26], [27]), and demonstrate its feasibility with a proof-of-concept implementation. This attack, which has a small but non-negligible attack surface, exploits certain IPv4 fragmentation policies used by the server and client operating systems (Section VI-E), rather than specific issues with NTP.

Network measurements (Sections III-B,V-F,VI-G-VI-H). The last measurement studies of the NTP ecosystem were conducted in 1999 [45] and 2006 [48], while a more recent study [14] focused on NTP DoS amplification attacks. We

study the integrity of the NTP ecosystem using data from the openNTPproject [39], and new network-wide scans (Section III-B). We identify bad timekeepers that could be exploited by off-path attacker (Section V-F), and servers that are vulnerable to our fragmentation attack (Sections VI-G-VI-H).

Recommendations and disclosure (Sections V-G,VI-I,VIII). Disclosure of these results began on August 20, 2015, and the Network Time Foundation, NTPsec, Redhat's security team, and Cisco quickly responded with patches to their NTP implementations. We have also worked with the openNTPproject to provide a resource that that operators can use to measure their servers' vulnerability to our fragmentation attacks.[1] Our recommendations for hardening NTP are in Sections IV-C,V-G,VI-I and summarized in Section VIII.

II. Why time matters: Implications of attacks on NTP

NTP lurks in the background of many systems; when NTP fails on the system, multiple applications on the system can fail, all at the same time. Such failures have happened. On November 19, 2012 [8], for example, two important NTP (stratum 1) servers, tick.usno.navy.mil and tock.usno.navy.mil, went back in time by about 12 years, causing outages at a variety of devices including Active Directory (AD) authentication servers, PBXs and routers [47]. Exploits of individual NTP clients also serve as a building block for other attacks, as summarized in Table I. Consider the following:

TLS Certificates. Several authors [44, pg 33, pg 183] [29], [62] have observed that NTP could be used to undermine the security of TLS certificates, which are used to establish secure encrypted and authenticated connections. An NTP attacker that sends a client back in time could cause the host to accept certificates that the attacker fraudulently issued (that allow the attacker to decrypt the connection), and have since been revoked[2]. (For example, the client can be rolled back to mid-2014, when $> 100K$ certificates were revoked due to heartbleed [71].) Alternatively, an attacker can send the client back to a time when a certificate for a cryptographically-weak key was still valid. (For example, to 2008, when a bug in Debian OpenSSL caused thousands of certificates to be issued for keys with only 15-17 bits of entropy [19].) Moreover, most browsers today accept (non-root) certificates for 1024-bit RSA keys, even though sources speculate that they can be cracked by well-funded adversaries [7]; thus, even a domain that revokes its old 1024-bit RSA certificates (or lets them expire) is vulnerable to cryptanalytic attacks when its clients are rolled back to a time when these certificates were valid. Some of these attacks were demonstrated by Selvi [62].

DNSSEC. DNSSEC provides cryptographic authentication of the Domain Name System (DNS) data. NTP can be used to attack a DNS resolver that performs 'strict' DNSSEC validation, *i.e.,* fails to return responses to queries that fail cryptographic DNSSEC validation. An NTP attack that sends a resolver forwards in time will cause all timestamps on DNSSEC cryptographic keys and signatures to expire (the recommended lifetime for zone-signing keys in DNSSEC is 1 month [33]); the resolver and all its clients thus lose connectivity to any domain secured with DNSSEC. Alternatively, an NTP attack that sends a resolver back in time allows for DNSSEC replay attacks; the attacker, for example, roll to a time in which a certain DNSSEC record for a domain name did not exist, causing the resolver to lose connectivity to that domain. Since the recommended lifetime for DNSSEC signatures is no more 30 days [33], this attack would need to send the resolver back time by a month (or more, if the time in which the DNSSEC record did not exist was further in the past).

To attack...	change time by ...	To attack...	change time by ...
TLS Certs	years	Routing (RPKI)	days
HSTS (see [61])	a year	Bitcoin (see [13])	hours
DNSSEC	months	API authentication	minutes
DNS Caches	days	Kerberos	minutes

TABLE I. Attacking various applications with NTP.

Cache-flushing attacks. NTP can also be used for cache flushing. The DNS, for example, relies heavily on caching to minimize the number of DNS queries a resolver makes to a public nameserver, thus limiting load on the network. DNS cache entries typically live for around 24 hours, so rolling a resolver forward in time by a day would cause most of its cache entries to expire. A widespread NTP failure (like the one in November 2012) could cause multiple resolvers to flush their caches all at once, simultaneously flooding the network with DNS queries [29], [44].

Interdomain routing. NTP can be used to exploit the Resource Public Key Infrastructure (RPKI) [36], a new infrastructure for securing routing with BGP. The RPKI uses Route Origin Authorizations (ROAs) to cryptographically authenticate the allocation of IP address blocks to networks. ROAs prevent hijackers from announcing routes to IP addresses that are not allocated to their networks. If a valid ROA is missing, a 'relying party' (that relies on the RPKI to make routing decisions) can lose connectivity to the IPs in the missing ROA.[3] As such, relying parties must always download a complete set of valid ROAs; to do this, they verify that they have downloaded all the files listed in cryptographically-signed 'manifest' files. To prevent the relying party from rolling back to a stale manifest that might be missing a ROA, manifests have monotonically-increasing 'manifest-numbers', and typically expire within a day [25]. NTP attacks, however, can first roll the relying party forward in time, flushing its cache and causing it to 'forget' its current manifest-number, and then roll the relying party back in time, so that it accepts a stale manifest as valid.

Bitcoin. Bitcoin is a digital currency that allows a decentralized network of node to arrive at a consensus on a distributed public ledger of transactions, *aka* "the blockchain". The blockchain consists of timestamped "blocks"; bitcoin nodes use computational proofs-of-work to add blocks to the blockchain. Because blocks should be added to the blockchain according to their validity interval (about 2 hours), an NTP attacker can trick a victim into rejecting a legitimate block, or

[1] http://www.cs.bu.edu/~goldbe/NTPattack.html

[2] The attacker must also circumvent certificate revocation mechanisms, but several authors [28], [34], [49] point out that this is relatively easy to do in various settings. For instance, several major browsers rely on OCSP [59] to check if a certificate was revoked, and default to "soft-fail", *i.e.,* accepting the certificate as valid, when they cannot connect to the OCSP server. NTP-based cache-flushing could also be useful for this purpose, by causing the client to 'forget' any old certificate revocation lists (CRLs) that it may have seen in the past; see also our discussion of routing attacks.

[3] See [12, Side Effect 6]: the relying party loses connectivity if it uses 'drop invalid' routing policy [12, Sec. 5], and the missing ROA has 'covering ROA'.

into wasting computational power on a stale block [13].

Authentication. Various services (*e.g.,* Amazon S3 [4], the DropBox Core API [16]) expose APIs that require authentication each time an application queries them. To prevent replay attacks, queries require a timestamp that is within some short window of the server's local time, see *e.g.,* [24, Sec 3.3]; Amazon S3, for example, uses a 15-minute window. Moreover, authentication with Kerberos requires clients to present freshly timestamped (typically within minutes) tickets to a server before being granted them access [32]. Thus, by changing a application's or server's time, an NTP attacker can deny service or launch replay attacks on various authentication services.

III. The NTP Ecosystem

We start with background on the NTP protocol, and use a measurement study to discuss its structure and topology. While NTP is one of the Internet's oldest protocols, it has evolved in more fluid fashion than other protocols like DNS or BGP. Thus, while NTP is described in RFC 5905 [43], practically speaking, the protocol is determined by the NTP reference implementation *ntpd*, which has changed frequently over the last decades [67]. (For example, *root distance* Λ (equation (4)) is a fundamental NTP parameter, but is defined differently in RFC 5905 [43, Appendix A.5.5.2], ntpd v4.2.6 (the second most popular version of ntpd that we saw in the wild) and ntpd v4.2.8 (the latest version as of May 2015).)

A. Background: The NTP Protocol.

NTP most commonly operates in an hierarchical client-server fashion.[4] Clients send queries to solicit timing information from a set of servers. This set of servers is manually configured before the client initializes and remains static over time. In general, the ntpd client can be configured with up to 10 servers.[5] Online resources suggest configuring anywhere from three to five servers [31], and certain OSes (*e.g.,* MAC OS X 10.9.5) default to installing ntpd with exactly one server (*i.e.,* time.apple.com). At the root of the NTP hierarchy are *stratum 1* NTP servers, that provide timing information to stratum 2 client systems. Stratum 2 systems provide time to stratum 3 systems, and so on, until stratum 15. Stratums 0 and 16 indicate that a system is unsynchronized. NTP servers with high stratum often provide time to the Internet at large (*e.g.,* pool.ntp.org, tick.usno.navy.mil); our organization, for example, has stratum 2 servers that provide time to internal stratum 3 machines, and take time from public stratum 1 servers.

Client/server communications. An NTP client and server periodically exchange a pair of messages; the client sends the server a *mode 3* NTP query and the server responds with a *mode 4* NTP response. This two-message exchange uses the IPv4 packet shown in Figure 1, and induces the following four important timestamps on the mode 4 response:

T_1 *Origin timestamp.* Client's system time when client sent mode 3 query.

[4]NTP also supports less popular modes: *broadcast*, where a set of clients are pre-configured to listen to a server that broadcasts timing information, and *symmetric peering*, where servers (typically at the same stratum) exchange time information. This paper just considers client-server mode.

[5]For example, when installing NTP in 14.04.1-Ubuntu in July 2015, the OS defaulted to installing ntpd v4.2.6 with a five preconfigured servers.

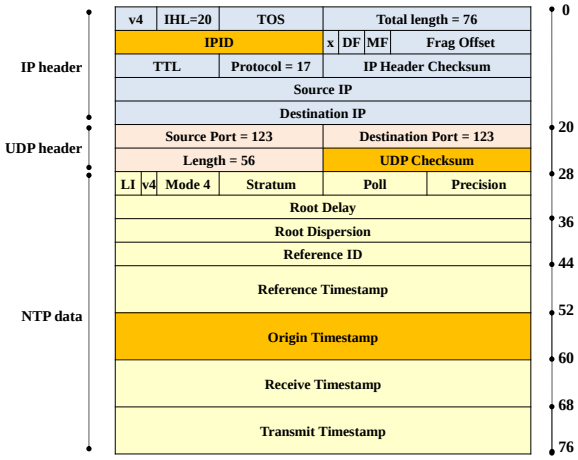

Fig. 1. Mode 4 NTP Packet, highlighting nonces and checksums.

T_2 *Receive timestamp.* Servers's system time when server received mode 3 query.

T_3 *Transmit timestamp.* Servers's system time when server sent mode 4 response.

T_4 *Destination timestamp.* Client's system time when client received mode 4 response. (Not in packet.)

The round-trip *delay* δ during the exchange is therefore:

$$\delta = (T_4 - T_1) - (T_3 - T_2) \tag{1}$$

Offset θ quantifies the time shift between a client's clock and a server's clock. Assume that delays on the forward (client→server) and reverse (server→client) network paths are symmetric and equal to $\frac{\delta}{2}$. Then, the gap between the server and client clock is $T_2 - (T_1 + \frac{\delta}{2})$ for the mode 3 query, and $T_3 - (T_4 - \frac{\delta}{2})$ for the mode 4 response. Averaging these two quantities gives:

$$\theta = \tfrac{1}{2}\left((T_2 - T_1) + (T_3 - T_4)\right) \tag{2}$$

An NTP client adaptively and infrequently selects a *single* server (from its set of pre-configured servers) from which it will take time. The IPv4 address of the selected server is recorded in the *reference ID* field of every NTP packet a system sends, and the *reference timestamp* field records the last time it synchronized to its reference ID. Notice that this means that any client querying a server S_2 can identify exactly which IPv4 NTP server S_1 the server S_2 has used for synchronization. (Meanwhile, it is more difficult to identify IPv6 NTP servers; because reference ID is 32-bits long, 128-bit IPv6 addresses are first hashed and then truncated to 32-bits [43, pg 22]. To identify an IPv6 server one would need a dictionary attack.)

A client and server will exchange anywhere between eight to hundreds of messages before the client deigns to take time from the server; we describe some of the algorithms used to make this decision in Section V-E. Messages are exchanged at infrequent *polling intervals* that are adaptively determined by a complex, randomized *poll process* [43, Sec. 13].

Authentication. How does the client know that it's talking to its real NTP server and not to an attacker? While NTPv4 supports both symmetric and asymmetric cryptographic authentication, this is rarely used in practice. Symmetric cryptographic authentication appends an MD5 hash keyed with symmetric key

ntpd version	4.1.1	4.2.6	4.1.0	4.2.4	4.2.0	4.2.7	4.2.8	4.2.5	4.4.2
# servers	1,984,571	702,049	216,431	132,164	100,689	38,879	35,647	20,745	15,901

TABLE II. TOP NTPD VERSIONS IN *rv* DATA FROM MAY 2015.

kernel	2.6.18	2.4.23	2.6.32	2.4.20	2.6.19	2.4.18	2.6.27	2.6.36	2.2.13
# servers	123,780	108,828	97,168	90,025	71,581	68,583	61,301	45,055	29550

TABLE IV. TOP LINUX KERNELS IN *rv* DATA FROM MAY 2015.

OS	Unix	Cisco	Linux	BSD	Junos	Sun	Darwin	Vmkernal	Windows
# servers	1,820,957	1,602,993	835,779	38,188	12,779	6,021	3625	1994	1929

TABLE III. TOP OSES IN *rv* DATA FROM MAY 2015.

stratum	0,16	1	2	3	4	5	6	7-10	11-15
# servers	3,176,142	115,357	1,947,776	5,354,922	1,277,942	615,633	162,162	218,370	187,348

TABLE V. STRATUM DISTRIBUTION IN OUR DATASET.

k of the NTP packet contents m as MD5$(k||m)$ [44, pg 264] to the NTP packet in Figure 1.The symmetric key must be pre-configured manually, which makes this solution quite cumbersome for public servers that must accept queries from arbitrary clients. (NIST operates important public stratum 1 servers and distributes symmetric keys only to users that register, once per year, via US mail or facsimile [3]; the US Naval Office does something similar [2].) Asymmetric cryptographic authentication is provided by the Autokey protocol, described in RFC 5906 [23]. RFC 5906 is not a standards-track document (it is classified as 'Informational'), NTP clients do not request Autokey associations by default [1], and many public NTP servers do not support Autokey (*e.g.,* the NIST timeservers [3], many servers in pool.ntp.org, and the US Naval Office (USNO) servers). In fact, a lead developer of the ntpd client wrote in 2015 [65]: "Nobody should be using autokey. Or from the other direction, if you are using autokey you should stop using it. If you think you need to use it please email me and tell me your story." For the remainder of this paper, we shall assume that NTP messages are unauthenticated.

B. Measuring the NTP ecosystem.

We briefly discuss the status of today's NTP ecosystem. Our measurement study starts by discovering IP addresses of NTP servers in the wild. We ran a zmap [18] scan of the IPv4 address space using mode 3 NTP queries on April 12-22, 2015, obtaining mode 4 responses from 10,110,131 IPs.[6] We augmented our data with openNTPproject [39] data from January-May 2015, which runs weekly scans to determine which IPs respond to NTP control queries. (These scans are designed to identify potential DDoS amplifiers that send large packets in response to short control queries [14].) The openNTPproject logs responses to NTP *read variable (rv)* control queries. *rv* responses provide a trove of useful information including: the server's OS (also useful for OS fingerprinting!), its ntpd version, its reference ID, the offset θ between its time and that of of its reference ID, and more. Merging our zmap data with the openNTPproject *rv* data gave a total of 11,728,656 IPs that potentially run NTP servers.

OSes and clients in the wild. We use openNTPproject's *rv* data to get a sense of the OSes and ntpd clients that are present in the wild. Importantly, the *rv* data is incomplete; *rv* queries may be dropped by firewalls and other middleboxes. Many NTP clients are also configured to refuse to respond to these queries, and some *rv* responses omit information. (This is why we had only 4M IPs in the *rv* data, while 10M IPs responded to our mode 3 zmap scan.) Nevertheless, we get some sense of what systems are out there by looking at the set of *rv* responses from May 2015. In terms of operating systems, Table III shows many servers running Unix, Cisco or Linux. Table IV indicates that Linux kernels are commonly v2 (rather the more recent v3); in fact, Linux v3.0.8 was only the 13^{th} most popular Linux kernel, with 17,412 servers. Meanwhile, Table II shows that ntpd v4.1.1 (released in 2001) and v4.2.6 (released in 2008) are most popular; the current release v4.2.8 (2014) is ranked only 8^{th} amongst the systems we see. The bottom line is that there are plenty of legacy NTP systems in the wild. As such, our lab experiments and attacks study the behavior of *two* NTP's reference implementations: ntpd v4.2.6p5 (the second most popular version in our dataset) and ntpd v4.2.8p2 (the latest release as of May 2015).

Bad timekeepers. Next, we used our mode 3 zmap data to determine how many *bad timekeepers*–servers that are unfit to provide time—are seen in the wild. To do this, we compute the offset θ (equation (2)) for each IP that responded to our mode 3 queries, taking T_1 from the Ethernet frame time of the mode 3 query, T_4 from the Ethernet frame time of the mode 4 query, and T_2 and T_3 from the mode 4 NTP payload. We found many bad timekeepers — 1.7M had $\theta \geq 10$ sec, 3.2M had stratum 0 or 16, and the union of both gives us a total of 3.7M bad timekeepers. Under normal conditions, NTP is great at discarding information from bad timekeepers, so it's unlikely that most of these servers are harming anyone other than themselves; we look into this in Sections V-D-V-F.

Topology. Since a system's reference ID reveals the server from which it takes time, our scans allowed us to start building a subset of the NTP's hierarchical client-server topology. However, a reference ID only provide information about *one* of a client's preconfigured servers. In an effort to learn more, on June 28-30, 2015 we used nmap to send an additional mode 3 NTP query to every IP that had only one parent server in our topology; merging this with our existing data gave us a total of 13,076,290 IPs that potentially run NTP servers. We also wanted to learn more about the clients that synchronize to bad timekeepers. Thus, on July 1, 2015, we used the openNTPproject's scanning infrastructure to send a monlist query to each of the 1.7M servers with $\theta > 10$ sec. While monlist responses are now deactivated by many servers, because they have been used in DDoS amplification attacks [14], we did obtain responses from 22,230 of these bad timekeepers. Monlist responses are a trove of information, listing all IPs that had recently sent NTP packets (of any mode) to the server. Extracting only the mode 3 and 4 data from each monlist response, and combining it with our existing data, gave us a total of 13,099,361 potential NTP servers.

Stratum. Table V shows the distribution of stratums in our entire dataset. Note that there is *not* a one-to-one mapping between an NTP client and its stratum; because a NTP client can be configured with servers of various stratum, the client's own stratum can change depending on the server it selects for synchronization. Thus, Table V presents the 'best' (*i.e.,*

[6]NTP control query scans run in 2014 as part of [14]'s research found several 'mega-amplifiers': NTP servers that response to a single query with millions of responses. Our mode 3 scan also found a handful of these.

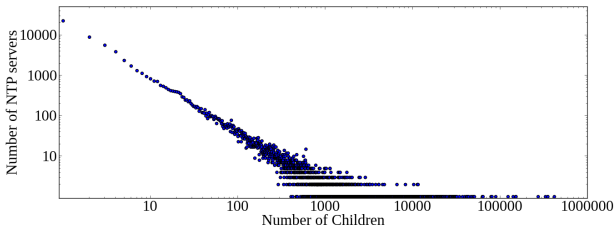

Fig. 2. Client-degree distribution of NTP servers in our dataset; we omit servers with no clients.

smallest) stratum for each IP in our dataset. Unsurprisingly, stratum 3 is most common, but, like [14] we also see many unsynchronized (stratum 0 or 16) servers.

Degree distribution. Figure 2 shows the client (*i.e.,* child) degree distribution of the servers in our topology. We note that our topology is highly incomplete; it excludes information about NTP clients behind a NAT or firewall, as well as servers that a client is configured for but not synchronized to.[7] The degree distribution is highly skewed. Of 13.1M IPs in our dataset, about 3.7M (27.8%) had clients below them in the NTP hierarchy. Of these 3.7M servers with clients, 99.4% of them have fewer than 10 clients, while only 0.2% of them have more than 100 clients. However, servers with more than 100 client tend to have many clients, averaging above 1.5K clients per server, with the top 50 servers having at least 24.5K clients each. Compromising these important servers (or hijacking their traffic) can therefore impact large swaths of the NTP ecosystem.

IV. How to step time with NTP.

Unauthenticated NTP traffic is vulnerable to on-path attacks, as was pointed out by various authors [23], [29], [46], [61], [62]. While on-path attacks are sometimes dismissed because the attacker requires a privileged position on the network, it is important to remember that an attacker can use various traffic hijacking techniques to place herself on the path to an NTP server. For instance, ntpd configuration files allow clients to name servers by either their IP or their hostname. (MAC OS X 10.9.5, for example, comes with an NTP client that is preconfigured to take time from the host time.apple.com, while many systems rely on the pool of NTP servers that share the hostname pool.ntp.org.) If the DNS entries for these hostnames are quietly hijacked [26], [27], then an attacker can quietly manipulate the NTP traffic they send. Moreover, NTP relies on the correctness of IP addresses; thus attacks on interdomain routing with BGP [21] (similar to those seen in the wild [15], [52]) can be used to divert NTP traffic to an attacker.

In Section II and Table I we saw that dramatic shifts in time (years, months) are required when NTP attacks are used inside larger, more nefarious attacks. Can an on-path attacker really cause NTP clients to accept such dramatic shifts in time?

[7]Earlier studies [45], [48] used monlist responses, which are now commonly deactivated, to obtain topologies. We present a new technique that exposes a client's servers in Section V-C, but as it is also a denial-of-service attack on the client, we have not used it to augment our measurements.

A. Time skimming

At first glance, the answer should be no. NTP defines a value called the *panic threshold* which is 1000 sec (about 16 minutes) by default; if NTP attempts to tell the client to alter its local clock by a value that exceeds the panic threshold, then the NTP client "SHOULD exit with a diagnostic message to the system log" [43]. Our experiments confirm that ntpd v4.2.6 and v4.2.8 quit when they are initially synchronized to a server that then starts to offer time that exceeds the panic threshold.

On way to circumvent this is through an adaption of [61]'s "time-skimming" technique,[8] so that the man-in-the-middle slowly steps the client's local clock back/forward in steps smaller than the panic threshold. However, this comes a big caveat: it can take minutes or hours for ntpd to update a client's local clock. To understand why, observe that in addition to the panic threshold, NTP also defines a *step threshold* of 125 ms [43]. A client will accept a time step larger than step threshold but smaller than the panic threshold as long as at least *"stepout"* seconds have elapsed since its last clock update; the *stepout* value is 900 seconds (15 minutes) in ntpd v4.2.6 and RFC 5905 [43], and was reduced to 300 seconds (5 minutes) in ntpd v4.2.8. Thus, shifting the client back one year using steps of size 16 minute each requires $\frac{1\times365\times24\times60}{16} = 33$K steps in total; with a 5 minute stepout value, this attack takes at least 114 days. However, there are other ways to quickly shift a client's time.

B. Exploiting reboot.

ntpd has a configuration option called `-g`, which allows an NTP client that first initializes (*i.e.,* before it has synchronized to any time source) to accept any time shift, even one exceeding the panic threshold. This configuration is quite natural for clocks that drift significantly when systems are powered down; indeed, many OSes, including Linux, run ntpd with `-g` by default. We have confirmed that both ntpd v4.2.6p5 and ntpd v4.2.8p2 on Ubuntu13.16.0-24-generic accept a single step 10 years back in time, and forward in time, upon reboot.

Reboot. An on-path attacker can exploit the `-g` configuration to dramatically shift time at the client by waiting until the client restarts as a result of power cycling, software updates, or other 'natural events'. Importantly, the on-path attacker knows exactly when the client has restarted, because the client puts 'INIT' in the reference ID of every NTP packet the client sends (Figure 1), including the mode 3 queries sent to the server. Moreover, the a determined attacker that can also use packet-of-death techniques (*e.g.,* Teardrop [9]) to deliberately reboot the OS, and cause ntpd to restart.

Feel free to panic. Suppose, on the other hand, that an NTP attacker shifts a client's time beyond the panic threshold, causing the client to quit. If the operating system is configured

[8]Selvi's [61] time-skimming attack allows for fast timesteps on time clients that update their local clock at predictable intervals—for instance, Fedora Linux sends a mode 3 query every minute and updates it's clock immediately upon receipt of the mode 4 response. The full ntpd implementation, however, has a significantly more complex clock update mechanisms. These mechanisms include (1) sending mode 3 queries at randomized and adaptively-selected intervals determined by the poll process [43, Sec 13], (2) only updating the clock upon receipt of an adaptively-chosen number of self-consistent mode 4 responses (see *e.g.,* the discussion of TEST11 in Section V-E), (3) using the stepout value to delay clock update events, *etc.*.

to reboot the NTP client, the rebooted NTP client will initialize and accept whatever (bogus) time it obtains from its NTP servers. Indeed, this seems to have happened with some OSes during the November 2012 NTP incident [40].

Small-step-big-step. Users might notice strange shifts in time if they occur immediately upon reboot. However, we have found that ntpd allows an on-path attacker to shift time when clients are less likely to notice.

To understand how, we need to look into ntpd's implementation of the `-g` configuration. One might expect `-g` to allow for timesteps that exceed the panic threshold only upon initialization—when the client updates its clock for the very first time upon starting. To implement this, the ntpd allows steps exceeding the panic threshold only when a variable called `allow_panic` is TRUE. It turns out that sets `allow_panic` to TRUE only upon initialization with the `-g` configuration (otherwise, it is initialized to FALSE), and is set to FALSE if the client (1) is just about to update its local clock by a value less than the step threshold (125ms), and (2) is already in a state called SYNC, which means it recently updated its clock by a value less than the step threshold. Normally, a client initializes and (1) and (2) occur after *two* clock updates. However, if an attacker is able to prevent the client from making ever two contiguous clock updates (one after the other) of less than 125 ms each, then `allow_panic` remains TRUE.

The following *small-step-big-step attack* on ntpd v4.2.6 exploits the above observation. First, the client reboots and begins initializing; it signals this to the server by putting 'INIT' in the reference ID of its mode 3 queries [43, Fig. 13]). Next, the client synchronizes to the server; it signals this with the server's IP address in the reference ID of its mode 3 queries [43, Fig. 13]). When the server sees that the client has synchronized once, the server sends the client a 'small step' greater than the STEP threshold (125 ms) and less than the panic threshold ($\approx$ 16 min); the client signals that it has accepted this timestep by putting 'STEP' in its reference ID [43, Fig. 13]). When the server sees that the client is in 'STEP' mode, the server immediately sends the client a big step that exceeds the panic threshold. At this point, the client does not panic, because it never set `allow_panic` to FALSE. Indeed, one might even interpret this as expected behavior per RFC 5905 [43]:

> STEP means the offset is less than the panic threshold, but greater than the step threshold STEPT (125 ms). In this case, the clock is stepped to the correct offset, but ... all associations MUST be reset and the client begins as at initial start.

Notice that this attack gives the server some 'slack time' before it sends the client the bogus big time step.

We confirmed this behavior with ntpd v4.2.6p5, with a small step of 10 minutes and a big step of 1 year; the client exchanges 3 packets with the server before it first synchronizes with it, then 21 packets before it accepts the small 10 minute step and gets into the STEP mode, and 25 packets before it accepts the big 1 year step and gets into STEP mode.

The small-step-big-step attack is slightly different with ntpd v4.2.8p2. With ntpd v4.2.8p2, `allow_panic` is set to FALSE under conditions (1) and (2), OR if (1) holds and (3) the client is in FSET state, which is the state the client enters upon initialization. Normally, a client initializes and (1) and (3) occur after *one* clock update. Thus, our small-step-big-step attacks works as long as *every clock update* the client receives exceeds the step threshold (125ms). We confirmed this behavior for an ntpd v4.2.8p2 client with a small step of 10 minutes and two big steps of 1 year. The very first mode 4 response received by the client, upon initialization, was the small step of 10 minutes back in time; the client immediately went into 'STEP' mode upon receipt of this packet. The next mode 4 response received by the client was a big step of 1 year back in time, which the client accepted after sending 11 queries to the server. The next mode 4 response was a another big step of 1 year, which the client accepted after sending 10 queries to the server.

Stealthy time shift. As an application of the small-step-big-step attack, an on-path attacker can preform a bogus big step and then quickly to bring the client's clock back to normal, so that the client never notices the time shift; this attack might be useful to stealthily flush a client's cache, or to cause a specific cryptographic object to expire (see Section II). To do this, the attacker waits for ntpd to reboot (or deliberately causes a reboot), and ensures that every clock update made by the client makes is larger than 125 ms, sending the client into STEP mode. To keep things stealthy, the attacker can *e.g.,* first shift the client forward in time by 150 ms, then back in time by 150 ms, then forward in time by 150 ms, *etc.*. Then, once the attacker is ready, it can send the client a big step that exceeds the panic threshold, perform its nefarious deeds, and finally send another big step that sets the client's clock back to the correct time.

C. Recommendation: Careful with `-g`

The security of ntpd should should not rely on 100% OS uptime, so users should be careful with the `-g` option. One solution is to not use the `-g` option. Alternatively, one can detect feel-free-to-panic attacks by monitoring the system log for panic events and being careful when automatically restarting ntpd after it quits. Monitoring should also be used detect suspicious reboots of the OS (that might indicate the presence of a small-step-big-step or other reboot-based on-path attacks). Implementors can prevent small-step-big-step attacks by patching ntpd to ensure that the `allow_panic` variable is set to FALSE after the very first clock update upon initialization; this issue has been captured in CVE-2015-5300. Moreover, implementors can prevent ntpd clients from putting 'INIT' in the reference ID of their NTP packets upon initializing; this would make it more difficult for on-path attackers to know when initialization is taking place, raising the bar for attacks that exploit reboot.

V. KISS-O'-DEATH: OFF-PATH DENIAL-OF-SERVICE ATTACKS.

In this section, we discuss how NTP security can be stymied by another aspect of the NTP protocol: the *'Kiss-o-death' (KoD)* packet. KoD packets are designed to reduce load at an NTP server by rate-limiting clients that query the server too frequently; upon receipt of a KoD packet from its server, the client refrains from querying that server for some period of

time [43, Sec 7.4]. We find that KoD packets can be trivially spoofed, even by an off-path attacker. We then show how an off-path attacker can send a single spoofed KoD packet and prevent a client from synchronizing to any of its preconfigured NTP servers for days or even years. We also consider using KoDs to pin a client to a bad timekeeper.

A. Why are off-path attacks hard?

We first need to understand why it is usually difficult to spoof NTP mode 4 packets (Figure 1) from off-path.

TEST2: The origin timestamp. Like many other protocols, NTP requires clients to check that a nonce in the server's query matches a nonce in the server's response; that way, an off-path attacker, that cannot observe client-server communications, does not know the nonce and thus has difficulty spoofing the packet. (This is analogous to source port randomization in TCP/UDP, sequence number randomization in TCP, and transaction ID randomization in DNS.) NTP uses the *origin timestamp* as a nonce: the client checks that (a) the origin timestamp on the mode 4 response sent from server to client (Figure 1), matches (b) the client's local time when he sent the corresponding mode 3 query, which is sent in the *transmit timestamp* field of the mode 3 query sent from client to server. This is called `TEST2` in the ntpd code. (Note that ntpd does not randomize the UDP source port to create an additional nonce; instead, all NTP packets have UDP source port 123.)

How much entropy is in NTP's nonce? The origin timestamp is a 64 bit value, where the first 32 bits represent seconds elapsed since January 1, 1900, and the last 32 bits represent fractional seconds. A client whose system clock has *e.g.,* $\rho = -12$ bit precision ($2^{-12} = 244\mu$s) puts a $32 - 12 = 20$-bit random value in the least significant bit of the timestamp. Thus, for precision ρ, the origin timestamp has at least $32 + \rho$ bits of entropy. However, because polling intervals are no shorter than 16 seconds [43], an off-path attacker is unlikely to know exactly when the client sent its mode 3 query. We therefore suppose that the origin timestamp has about 32 bits of entropy. This is a lot of entropy, so one might conclude that NTP is robust to off-path attacks. However, in this section and Section VI, we will show that this is not the case.

B. Exploiting the Kiss-O'-Death Packet

A server sends a client a Kiss-O'-Death (KoD) if a client queries it too many times within a specified time interval; the parameters for sending KoD are server dependent. A sample KoD packet is shown in Figure 3. The KoD is characterized by mode 4, leap indicator 3, stratum 0 and an ASCII 'kiss code' string in the reference ID field. According to RFC5905 [43]:

> For kiss code RATE, the client MUST immediately reduce its polling interval to that server and continue to reduce it each time it receives a RATE kiss code.

In ntpd v4.2.6 and v4.2.8, this is implemented by having the client stop querying the server for a period that is at least as long as the poll value field in the received KoD packet.[9] Our experiments confirm that if the KoD packet has polling interval $\tau_{\text{kod}} = 17$ (the maximum allowable polling interval [43]) then the ntpd v4.2.8 client will stop querying the server for at least $2^{\tau_{\text{kod}}}$ sec (36 hours). [10] The poll field in the NTP packet is an 8-bit value (*i.e.,* ranging from 0 to 255), but RFC 5905 [43, pg 10] defines the maximum allowable poll value to be 17. The most recent ntpd implementation, however, will accept KoDs with poll values even larger than 17; setting $\tau_{\text{kod}} = 25$, for example, should cause the client to stop querying its server for at least 2^{25} seconds, or about 1 year.

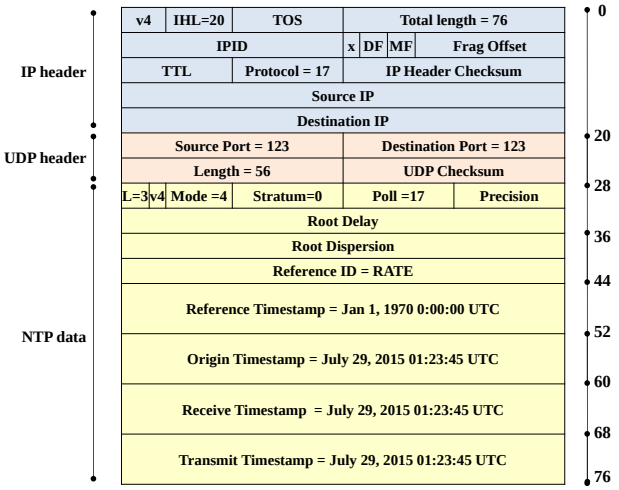

Fig. 3. Kiss-o'-Death (KoD) packet, telling the client to keep quiet for at least 2^{17} seconds (36 hours).

Spoofing a KoD from off-path. How does the client know that the KoD packet came from the legitimate server, and not from an attacker? With regular mode 4 responses, the client uses the origin timestamp as a nonce. While it seems reasonable to expect this check to be performed on the KoD packet as well, RFC 5905 [43, Sec. 7.4] does not seem to explicitly require this. Moreover, lab experiments with ntpd v4.2.6p5 and v4.2.8p3 show that the client accepts a KoD even if its origin timestamp is bogus. This means that an offpath attacker can trivially send the client a KoD that is spoofed to look like it came from its server; the only information the attacker needs is the IP addresses of the relevant client and server. Moreover, by setting the poll value in the spoofed KoD to be an unreasonably high value (*e.g.,* $\tau_{\text{kod}} = 25$), the spoofed KoD will prevent the client for querying its server for an extended period of time (*e.g.,* a year). This issue has been captured in CVE-2015-7704 and was patched, after the disclosure of our work, in ntpd v4.2.8p4.

[9] Interestingly, RFC 5905 [43, Sec. 7.4] defines an even more dangerous type of KoD packet: " For kiss codes DENY and RSTR, the client MUST demobilize any associations to that server and stop sending packets to that server". Thus, spoofing a single DENY or RSTR KoD packet can completely disconnect a client from its server! Fortunately, however, we have not found an implementation that honors this functionality.

[10] Due to complex interactions between τ_{kod}, the poll value in the KoD packet, and NTP's polling algorithm, the period of time that the client stops querying the server will usually exceed $2^{\tau_{\text{kod}}}$. More precisely, the client's polling algorithm resets the minpoll value $\tau_{\min}$ for the server sending the KoD to $\tau_{\min} = \max(\tau_{\min}, \tau_{\text{kod}})$ and then the client stops querying the server for a period of about $2^{\max(\tau_{\min}+1, \max(\min(\tau_{\max}, \tau_{\text{kod}})))}$. By default, minpoll is initialized to $\tau_{\min} = 6$, and maxpoll to $\tau_{\max} = 10$.

Eliciting a KoD from off-path: Priming the pump. Moreover, even if the client did validate the origin timestamp on the KoD packet, an off-path attacker could still *elicit* a valid KoD packet for the client from the server. The idea here is to have the off-path attacker 'prime-the-pump' at the server, by sending multiple mode 3 queries spoofed to look like they come from the victim client; the server then 'gets angry' at the client, and responds to the client's legitimate mode 3 queries with a KoD. The attacker just needs to measure the number of times q in a given period of time t_0 that a client must query the server in order to elicit a KoD; this is easily done by attempting to get the server to send KoD packets to the attacker's own machine. Then, the attacker sends the server $q-1$ queries with source IP spoofed to that of the victim client, and hopes that the client sends its own query to server before time t_0 elapses. If this happens, the server will send the client a KoD packet with valid origin timestamp (matching the query that the client actually sent to the server), and the client will accept the KoD. This issue has been captured in CVE-2015-7705. Interestingly, recent NTP security bulletins have increased this attack surface by recommending that servers send KoDs [5].[11]

Attack efficiency. The current ntpd implementation (v4.2.8p3) does not require clients to validate the origin timestamp on the KoD. This means that a *single spoofed KoD packet* with a unreasonably high poll value (*e.g.,* $\tau_{\text{kod}} = 25$) will essentially prevent the client from ever taking time from its server. Meanwhile, even if the client does validate the KoD's origin timestamp, an off-path attacker can still elicit a valid KoD by priming the pump. This, however, requires the attacker to expend more resources by sending more packets to the client's server. Specifically:

1) The attacker must send several packets to the server to elicit the KoD, and

2) The attacker no longer controls the poll value in the elicited KoD packet. When ntpd servers send KoDs, the KoD's poll value τ_{kod} is at least as large as that in the query triggering the KoD. Since (a) the elicited KoD is sent in response to a legitimate query sent by the client (so that it has a valid origin timestamp), and (b) the poll value in the client's query should not exceed $\tau = 10$ (the default maxpoll), it follows that the elicited KoD packet is likely to have poll value no larger than 10. The attacker can thus elicit a KoD that keeps the client quiet for about 2^{10} seconds (15 minutes), prime-the-pump again 15 minutes later to elicit another KoD, and continue this indefinitely.

Thus, requiring clients to validate the origin timestamp on the KoD will weaken, but not eliminate, any KoD-related attacks.

C. Low-rate off-path denial-of-service attack on NTP clients.

Its tempting to argue that NTP clients are commonly pre-configured with multiple servers, and thus any KoD-related attacks on one server can be mitigated by the presence of the other servers. However, this is not the case.

We present a denial-of-service attack that allows an off-path attacker, located anywhere on the Internet, to "turn off" NTP at that client by preventing the client from synchronizing to any of its preconfigured servers. What are the implications of this attack? For the most part, the client will just sit there and rely on its own local clock for the time. If the client has accurate local time-keeping abilities, then this attack is unlikely to cause much damage. On the other hand, the client machine could be incapable of keeping time for itself, *e.g.,* because it is in a virtual machine [70], or running CPU-intensive operations that induce clock drift. In this case, the client's clock will drift along, uncorrected by NTP, for the duration of attack.

The denial of service attack. The attack proceeds as follows:

1) The attacker sends a mode 3 NTP query to the victim client, and the client replies with a mode 4 NTP response. The attacker uses the reference ID in the mode 4 response to learn the IP of the server to which the client is synchronized.

2) The attacker spoofs/elicits a KoD packet with τ_{kod} from the server to which the client is synchronized. The client stops querying this server for at least $2^{\tau_{\text{kod}}}$ sec.

3) There are now two cases. Case 1: the client declines to take time from any its of other preconfigured servers; thus, the attacker has succeeded in deactivating NTP at the client. Case 2: The client will synchronize to another one of its preconfigured servers, and the attacker returns to step 1. To determine whether the client is in the Case 1 or Case 2, the attacker periodically sends mode 3 queries to the client, and checks if the reference ID in the mode 4 response has changed.

Thus, the attacker learns the IP addresses of all the preconfigured servers from which the client is willing to take time, and periodically (*e.g.,* once every $2^{\tau_{\text{kod}}}$ seconds), spoofs KoD packets from each of them. The client will not synchronize to any of its preconfigured servers, and NTP is deactivated. Moreover, this attack can continue indefinitely.

Attack surface. For this attack to work, the client must (1) react to KoD packets by refraining from querying the KoD-sending server, (2) respond to NTP mode 3 queries with NTP mode 4 responses, and (3) be synchronized an IPv4 NTP server. This creates a large attack surface: condition (1) holds for ntpd v4.2.6 and v4.2.8p3, the most recent reference implementation of NTP, and our scans (Section III-B) suggest that over 13M IPs satisfy condition (2).

Sample experiment. We ran this attack on an ntpd v4.2.8p2 client in our lab configured with the IP addresses of three NTP servers in the wild. We elicited a KoD for each server in turn, waiting for the client that resynchronize to a new server before eliciting a new KoD from that server. To elicit a KoD, a (separate) attack machine in our lab ran a scapy script that sent the server 90 mode 3 queries in rapid succession, each of which was spoofed with the source IP of our victim client and origin timestamp equal to the current time on the attacker's machine. (Notice that the origin timestamp is bogus from the perspective of the victim client.) Because our spoofed mode 3 queries had poll value $\tau = 17$, the elicited KoDs had poll value $\tau = 17$. Our client received its third KoD within 1.5 hours, and stopped querying its servers for the requested 2^{17} seconds (36 hours); in fact, the client had been quiet for 50 hours when we stopped the experiment.

[11] As of August 2015, [5] recommends the configuration `restrict default limited kod nomodify notrap nopeer`. Note that turning on `limited` means that a server will not serve time to a client that queries it too frequently; `kod` additionally configures the server to send KoDs.

D. Pinning to a bad timekeeper. (Part 1: The attack)

Consider a client that is preconfigured with several servers, one of which is a bad timekeeper. Can KoDs force the client to synchronize to the bad timekeeper? Indeed, since ntpd clients and configurations are rarely updated (see Table II, that shows that 1.9M servers use a version of ntpd that was released in 2001), a bad timekeeper might be lurking in rarely-updated lists of preconfigured servers.

The attack. The attacker uses the KoD denial-of-service attack (Section V-C) to learn the client's preconfigured servers; each time the attacker learns a new server, the attacker sends a mode 3 query to the server to check if it is a good timekeeper, continuing to spoof KoDs until it identifies a server that is a bad timekeeper. At this point, the client is taking time from a bad timekeeper, and the attack succeeds.

But does this attack actually work? NTP's clock discipline algorithms are designed to guard against attacks of this type. We have launched this attack against different clients (in our lab) configured to servers (in the wild), and observed differing results; sometimes, that client does take time from the bad timekeeper, and sometimes it does not. (See the full version for details on our experiments.) Unfortunately, however, we have not yet understood exactly what conditions are required for this attack to succeed. One thing we do know, however, are conditions that would definitely cause this attack to fail. We explain these conditions by taking detour into some aspects of NTP's clock discipline algorithm, that will also be important for the attacks in Section VI.

E. Detour: NTP's clock discipline algorithms.

An NTP client only updates its local clock from its chosen server at infrequent intervals. Each valid mode 4 response that the client obtains from its server is a *sample* of the timing information at that server. A client needs to obtain enough "good" samples from a server before it even considers taking time from that server. Empty samples are recorded under various failure conditions; we discussed one such failure condition, `TEST2` (origin timestamp validity) in Section V-A.

For each non-empty sample, the client records the offset θ per equation (2) and delay δ per equation (1). The client keeps up to eight samples from each server, and selects the offset θ^* corresponding to the non-empty sample of lowest delay δ^*. It then computes the *jitter* ψ, which is the root-mean-square distance of the sample offsets from θ^*, *i.e.,*

$$\psi = \sqrt{\frac{1}{i-1}\sum_i (\theta_i - \theta^*)^2} \tag{3}$$

Next, the server must pass another crucial check:

`TEST11`. Check that the *root distance* Λ does not exceed MAXDIST, a parameter that defaults to 1.5 sec. While RFC 5905 [43, Appendix A.5.5.2], ntpd v4.2.6 and ntpd v4.2.8 each use a slightly different definition of Λ, what matters is

$$\Lambda \propto \psi + (\delta^* + \Delta)/2 + E + 2^\rho \tag{4}$$

where Δ is the *root delay*, E is the *root dispersion*, and ρ is the *precision*, all which are read off the server's mode 4 packet per Figure 1. (Precision ρ reveals the resolution of the server's local clock; *e.g.,* $\rho = -12$ means the server's local clock is precise to within 2^{-12} sec or 244 μs. Root delay Δ is the cumulative delay from the server to the 'root' (*i.e.,* stratum 1 server) in the NTP client-server hierarchy from which the server is taking time; a stratum 1 server typically has $\Delta = 0$. Root dispersion E is an implementation-dependent quantity related to the Λ value computed by the server.)

If the client has several servers that pass `TEST11` (and other tests we omit here), the client must select a single server to synchronize its clock. This is done with a variant of Marzullo's Algorithm [38], which clusters servers according to offset θ^* (equation (2)) and other metrics. Servers who differ too much from the majority are ignored, while the remaining servers are added to a 'survivor list'. (This is why, under normal conditions, NTP clients do not synchronize to bad timekeepers.) If all servers on the survivor list are *different* from the server the client used for its most recent clock update, the client must decide whether or not to "*clock hop*" to a new server. The clock hop decision is made by various other algorithms that are different in ntpd v4.2.6 and ntpd v4.2.8. If no clock hop occurs, the client does not update its clock.

F. Pinning to a bad timekeeper. (Part 2: Attack surface)

Thus, for a client to synchronize to a bad timekeeper, we know that the bad timekeeper must be able to pass `TEST11`. In practice, we found that this means that bad timekeeper must send mode 4 packets (Figure 1) with root delay Δ and root dispersion E and precision ρ such that $\Delta/2 + E + 2^\rho < 1$ sec. In addition to this, the bad timekeeper must (1) "defeat" the good timekeepers in Marzullo's algorithm, and then (2) convince the client to clock hop. As part of ongoing work, we are attempting to determine the requirements for (1) and (2).

KoD + reboot. There is one way around all of the issues discussed so far. If ntpd reboots,[12] then the attacker has a short window during which the client is not synchronized to *any* of its preconfigured servers; our experiments confirm that in ntpd v4.2.6 this window is at least 4 polling intervals (*i.e.,* the minimum polling interval is 2^4 sec, so this translates to at least 1 min), while ntpd v4.2.8 shortens this to after the client receives its first mode 4 responses from its servers. Thus, the attacker can exploit this short window of time to spoof a KoD packet from each of the clients' good-timekeeper servers, but not from the bad timekeeper. As long as the bad timekeeper passes `TEST11`, the attacker no longer has to worry about Marzullo's algorithm, clock-hopping, or exceeding the panic threshold, and the attack will succeed.

Bad timekeepers in the wild. To quantify the attack surface resulting from pinning to a bad timekeeper, we checked which bad timekeepers (with offset $\theta > 10$ sec) are "good enough" to have (a) $\Delta/2 + E + 2^\rho < 1$ sec and (b) stratum that is not 0 or 16. Of the 3.7M bad timekeepers in our dataset (Section III-B), about 368K (or 10%) are "good enough" to pass the necessary tests. Meanwhile, we found only 2190 bad timekeepers that had clients below them in the NTP hierarchy, and of these, 743 were "good enough". While our topology is necessarily

[12] A reboot can be elicited by sending a packet-of-death that restarts the OS (*e.g.,* Teardrop [9]) or the ntpd client (*e.g.,* CVE-2014-9295 [50]). Alternatively, the attacker can wait until the client reboots due to a power cycling, software updates, or other 'natural' events.

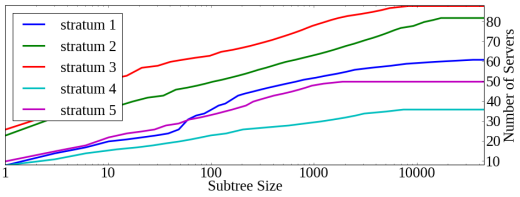

Fig. 4. Cumulative distribution of the size of the subtrees rooted at bad timekeepers (with offset $\theta > 10$ sec) that can pass `TEST11` (because $\Delta/2 + E + 2^{\rho} < 1$ sec), broken out by the bad timekeeper's stratum. We omit bad timekeepers with no children.

incomplete (Section III-B), this suggests that there are only a limited number of servers in the wild that can be used for these attacks. Figure 4 is a cumulative distribution of the size of the subtrees rooted at bad timekeepers that are "good enough" to pass the necessary tests, broken out by the bad timekeeper's stratum. Notice, however, that these distributions are highly skewed, so some of the "good enough" bad timekeepers had large subtrees below them in the NTP hierarchy. For example, one stratum 1 server (in Central Europe) had an offset of 22 minutes and almost 20K clients in its subtree.

G. Recommendation: Kiss-o'-Death considered harmful.

Many of the problems we have described here arise because KoD packets can be trivially spoofed. When we disclosed this paper to the Network Time Foundation, NTPsec and others, they quickly updated their implementations to validate the origin timestamp on the KoD (Section V-A). Thus, a KoD is no longer trivially spoofable with ntpd v4.2.8p4.

But does validating the origin timestamp on the KoD eliminate the problems we have discussed here? While this is certainly an improvement on the current state of affairs, we still contend that the origin timestamp is not a sufficient defense against KoD-related security issues. As we argued in Section V-B, even if clients validate the origin timestamp in the KoD packet, an attacker can still elicit valid KoDs by 'priming the pump' at any server that is willing to send KoDs. Priming-the-pump, however, does require the attacker to send more packets, as compared to just sending a single spoofed KoD.

There are several ways to defend against attackers that 'prime the pump' to elicit KoDs. NTP could simply eliminate its KoD and other rate-limiting functionality; this, however, eliminates a server's ability to deal with heavy volumes of NTP traffic. Alternatively, if clients are required to cryptographically authenticate their queries to the server, then it is no longer possible for an off-path attacker to prime the pump at the server by spoofing mode 3 queries from the client. Interestingly, however, a new proposal for securing NTP [64, Sec. 6.1.3] only suggests authenticating mode 4 responses from the server to client, but not mode 3 queries from server to client. Alternatively, in the absence of authentication, NTP can apply techniques developed for rate-limiting other protocols, *e.g.,* Response Rate Limiting (RRL) in the DNS [68]. With RRL, nameservers do not respond to queries from clients that query them too frequently.[13] Like NTP, DNS is sent over unauthenticated UDP, and therefore is at risk for the same priming-the-pump attacks we discussed here. RRL addresses this by requiring a server to randomly respond to some fraction of the client's queries, even if that client is rate limited [69, Sec. 2.2.7]; thus, even a client that is subject to a priming-the-pump attack can still get some good information from the server. To apply this to NTP, a server that is rate-limiting a client with KoDs would send legitimate mode 4 responses (instead of a KoD) to the client's queries with some probability. For this to be effective, NTP clients should also limit the period for which they are willing to keep quiet upon receipt of a KoD; not querying the server for days ($\tau_{\text{kod}} = 17$) or even years ($\tau_{\text{kod}} = 25$) upon receipt of a single KoD packet is excessive and invites abuse.

[13] ntpd also offers this type of rate limiting, as an alternative to the KoD, via the `limited` configuration option

VI. Off-Path NTP Fragmentation Attack

In this section, we show how an *off-path* attacker can hijack an unauthenticated NTP connection from a client to its server. The key ingredient in our attack is IPv4 packet fragmentation; therefore this attack succeeds against clients and servers that use certain classes of IPv4 packet fragmentation policies (Section VI-E). We will assume the client is preconfigured with only one server; some OSes (*e.g.,* MAC OS X v10.9.5) actually do use this configuration, and the KoD + reboot technique from Section V-F can simulate this scenario for clients preconfigured with multiple servers. We first explain why off-path attacks are challenging, and then provide some background on IPv4 packet fragmentation [54] and overlapping IPv4 packet fragments [51], [56], [63]. Next, we present the attack itself, explain when it works, and conclude with a measurement study that sheds light on the number of clients and servers in the wild that are vulnerable to this attack.

A. Why are off-path attacks hard?

The goal of our attacker is to spoof a series of mode 4 response packets (Figure 1) from the server to the client. The spoofed response should contain bogus server timestamps (*i.e.,* T_3 transmit timestamp, T_2 receive timestamp) in order to convince the client to accept bogus time from the server. This creates several hurdles for an off-path attacker who cannot see the communication between client and server:

First, there is the issue of nonces. Per Section V-A, the attacker must spoof packets with the correct origin timestamp, which has about 32 bits of entropy. Our off-path attacker will not even try to learn the origin timestamp; instead, we use the origin timestamp from the honest mode 4 response from server to client, and use IPv4 packet fragmentation to overwrite other relevant fields of the NTP packet. Note that origin timestamp is the only nonce in the NTP packet; UDP source-port randomization is not used by any of the NTP clients we have tested, which set source port to 123.

Second, since our attacker does not know NTP's origin timestamp, it cannot compute the UDP checksum. However, the UDP specification for IPv4 allows a host to accept any packet with UDP checksum of zero (which means: don't bother checking the checksum) [53, pg 2]. As such, our attacker uses IPv4 fragmentation to set the UDP checksum to zero.

Third, in order to convince the client's clock discipline algorithms (Section V-E) to accept the attacker's bogus time, our attacker must spoof a *stream* of several (at least eight, but usually more) packets that are acceptable to the clock

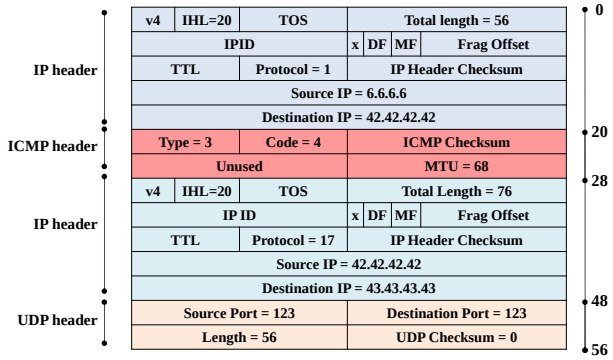

Fig. 5. ICMP Fragmentation Needed packet from attacker 6.6.6.6 telling server 42.42.42.42 to fragment NTP packets for client 43.43.43.43 to MTU of 68 bytes.

discipline algorithm. This is significantly more challenging than just spoofing a *single* packet as in *e.g.,* [26], [27]. Moreover, this stream of spoofed packets must be sufficiently self-consistent to pass `TEST11` (Section V-E), which, as we shall see, can be even more challenging.

B. IPv4 packet fragmentation.

Packet fragmentation is one of IP's original functionalities [54]; chopping a large packet into fragments that can be sent through networks that can only handle short packets. The length of the largest packet that a network element can accommodate is its '*maximum transmission unit (MTU)*'. In practice, almost every network element today supports an MTU of at least 1492 bytes (the maximum payload size for Ethernet v2 [37, Sec. 7]). Back in 1981, however, RFC791 [54] required that "all hosts must be prepared to accept" IPv4 packets of length 576 bytes, while "every internet module must be able to forward" IPv4 packets of length 68 bytes. The minimum IPv4 MTU for the Internet is therefore 68 bytes, but many OSes refuse to fragment packets to MTUs smaller than 576 bytes. Our attack only succeeds against servers that are willing to fragment to a 68 byte MTU; this way the attacker can convince a server to chop an NTP packet into the two fragments on the right of Figure 6. Our measurements (Section VI-G) confirm that there are ten of thousands of NTP servers in the wild that do this.

ICMP Fragmentation Needed. How does a host learn that it needs to fragment packets to a specific MTU? Any network element on the path from sender to receiver can send a single *ICMP fragmentation needed* packet to the sender containing the desired MTU; this information is then cached for some OS-dependent period of time (*e.g.,* 10 minutes by default on Linux 3.13.0 and MAC OS X 10.9.5). Figure 5 shows an *ICMP fragmentation needed* packet that signals to host 42.42.42.42 to fragment all NTP packets (UDP port 123) it sends to destination IP 43.43.43.43 to an MTU of 68 bytes. Since the host is not expected to know the IP addresses of all the network elements on its path, this packet can be sent from any source IP; in Figure 5 this source IP is 6.6.6.6. The payload of this ICMP packet contains an IP header and first eight bytes of a packet that has already been sent by host and exceeded the MTU [55]; for NTP, these eight bytes correspond to the UDP header. The sender uses this to determine which destination IP (*i.e.,* 43.43.43.43) and protocol (*i.e.,* UDP port 123) requires fragmentation. Our attacker (at IP 6.6.6.6) can easily signal an *ICMP fragmentation needed* from off-path. Its only challenge is (1) choosing UDP checksum (which it sets to zero) and (2) matching the IPID in the ICMP payload with that in an NTP packet previously sent to the client (which it can do, see Section VI-D, and moreover some OSes don't bother checking this (*e.g.,* Linux 3.13.0)).

IPv4 Fragmentation. How do we know that an IPv4 packet is a fragment? Three IPv4 fields are relevant (see Figure 1). *Fragment offset* specifies the offset of a particular fragment relative to the beginning of the original unfragmented IP packet; the first fragment has an offset of zero. The *more fragment (MF)* flag is set for every fragment except the last fragment. *IPID* indicates that a set of fragments all belong to the same original IP packet. Our attacker infers IPID (Section VI-D), and then sends the client spoofed IPv4 fragments with the same IPID as the legitimate fragments sent from the server, as in Figure 6. The spoofed and legitimate fragments are reassembled by the client into a single crafted NTP packet.

Fragment reassembly. How does a host reassemble a fragmented IPv4 packet? In the common case, the fragments are *non-overlapping*, so that the offset of one fragment begins immediately after the previous fragment ends. In this case, the host checks its *fragment buffer* for fragments with the same IPID, and pastes their payloads together according to their fragment offset, checking that the last fragment has a MF=0 [54]. Fragment buffer implementations differ in different OSes [26], [30]. Meanwhile, the RFCs are mostly silent about reassembly of *overlapping* fragments, like the ones in Figure 6.[14] Several authors [6], [51], [56], [63] have observed that reassembly policies differ for different operating systems, and have undertaken to determine these policies using clever network measurement tricks. (Hilariously, wireshark has an overlapping fragment reassembly policy that is independent of its host OS [6] and is therefore useless for this purpose.) Our attacks also rely on overlapping fragments. Overlapping fragment reassembly policies are surprisingly complex, poorly documented, and have changed over time. Thus, instead of generically describing them, we just consider reassembly for the specific fragments used in our attack.

C. Exploiting overlapping IPv4 fragments.

Our attack proceeds as follows. The attacker sends the server a spoofed *ICMP fragmentation needed* packet (Figure 5) requesting fragmentation to a 68-byte MTU for all NTP packets sent to the client. If the server is willing to fragment to a 68-byte MTU, the server sends all of its mode 4 NTP responses as the two fragments on the right of Figure 6. Meanwhile, our attacker plants the two spoofed fragments on the left of Figure 6 in the client's fragment buffer. The spoofed fragments sit in the fragment buffer and wait for the server's legitimate fragments to arrive. The first spoofed fragment should set the UDP checksum to zero while the second spoofed fragment should set the NTP receive timestamp (T_2) and transmit timestamps (T_3) to bogus time values. Both spoofed fragments must have the same IPID as the two legitimate fragments; we explain how to do this in Section VI-D. This

[14] RFC 3128 [42] does have some specific recommendations for overlapping IPv4 fragments in the case of TCP; NTP, however, is sent over UDP. Also, overlapping fragments are forbidden for IPv6.

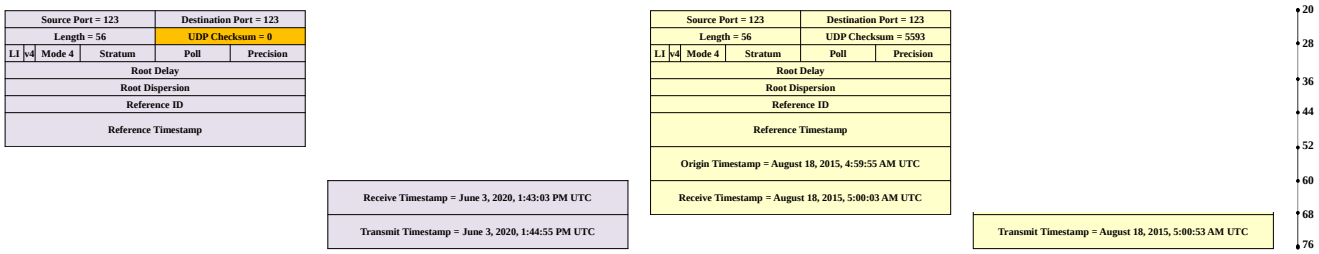

Fig. 6. IPv4 fragments for our attack: 1^{st} and 2^{nd} spoofed fragments followed by 1^{st} and 2^{nd} legitimate fragments.

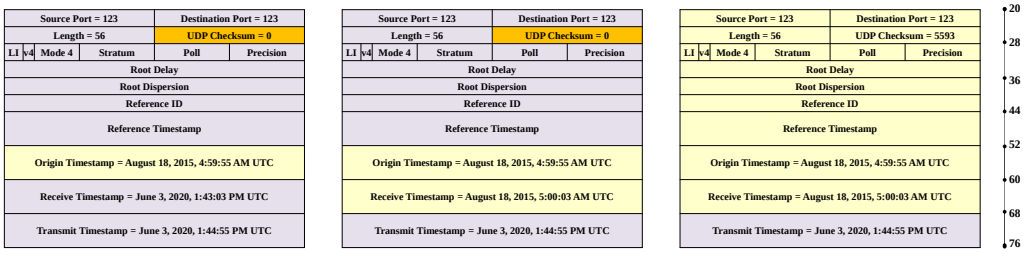

Fig. 7. Different ways our fragments may be reassembled. From left to right: Outcome A, Outcome B, Outcome C.

process of planting spoofed fragments continues for every mode 4 NTP response that the server sends the client. One the client has accepted the bogus time, the attacker spoofs KoDs (Section V-C) so the client stops updating its clock. (The attacker can check that the client accepted the bogus time by sending it a mode 3 queries and checking the timestamps in the client's mode 4 response.)

The victim client receives the four overlapping fragments in Figure 6, in the order shown, with the leftmost fragment arriving earliest. How are they reassembled? One potential outcome is for the client to reject the fragments altogether because they are overlapping or too short. Otherwise, the first honest fragment arrives in the client's fragment buffer and is reassembled with one or both of the spoofed fragments, according to one of the reassembly outcomes shown in Figure 7. In Outcome A the client prefers fragments that arrive earliest, pasting the first legitimate fragment underneath the two spoofed fragments that were waiting in the cache (*i.e.,* the 'First' policy in the Paxson/Shankar overlapping-packet reassembly model [63], and the 'First', 'Windows' and 'Solaris' policies of Novak [51]). In Outcome B, the client prefers an earlier fragment with an offset that is less than or equal to a subsequent fragment (*i.e.,* the 'BSD' policy of [51], [63]). In Outcome C the client prefers fragments that arrive later over those that arrive earlier (*i.e.,* the 'Last' and 'Linux' policies of [51], [63].

In which outcome does our attack succeed? In Outcome C, the packet is dropped due to its incorrect UDP checksum, and our attack fails. In Outcomes A and B, our off-path attacker successfully injects packets with the correct origin timestamp and UDP checksum. Both Outcome A and B also allow the attacker to control all the fields in NTP packet up to the reference timestamp, including stratum, precision ρ, root delay Δ, root dispersion E, and polling interval τ_p. This is useful because our attacker must ensure that the reassembled packets pass `TEST11`, so that root distance $\Lambda < 1.5$ (see Section V-E and equation (4)). Thus, our attacker can set these fields to tiny values in the first spoofed fragment, *e.g.,* $\rho = -29$, $\Delta = 0.002$, $E = 0.003$ sec and stratum = 1. Moreover, in Outcome A the attacker controls *both* the NTP transmit timestamp T_3 and receive timestamp T_2; by setting $T_2 \approx T_3$, the delay δ (equation (1)) is small enough to pass `TEST11`, even if when spoofed T_2 and T_3 are very far from the legitimate time. Meanwhile, in Outcome B, the attacker controls only the transmit timestamp T_3; passing `TEST11` constrains the spoofed T_3 to be within about 1 sec of the legitimate T_2. Thus in Outcome B, the attacker can only shift time by 1 sec, making the attack less interesting. Our attack works best when the client reassembles fragments as in Outcome A.

D. Planting spoofed fragments in the fragment buffer.

Because a client will only take time from a server that provides several self-consistent time samples (Section V-E), our attacker must craft a *stream* of NTP mode 4 responses. In achieving this, our attacker must surmount two key hurdles:

Hurdle 1: Jitter. Our attacker must ensure that the reassembled *stream* of packets is sufficiently self-consistent to pass `TEST11`, so that root distance $\Lambda < 1.5$ sec (equation (4)). We already discussed (Section VI-C) how the attacker can choose tiny values for the precision ρ, root delay Δ and root dispersion E and delay δ . The remaining difficulty is therefore the jitter ψ. Jitter ψ (equation (3)) is the variance in the offset values θ_i (equation (2)) computed from packets in the stream. To pass `TEST11`, the offset values θ in the reassembled stream of packets must be consistent to within about 1 sec.

Why is this difficult? The key problem is that the offset θ is determined by the timestamps T_2 and T_3 set in the attacker's second spoofed fragment (Figure 6), as well as the origin and destination timestamps T_1, T_4. T_1 corresponds to the moment when the legitimate client sends its query, and is unknown to our off-path attacker. Moreover, T_1 roughly determines T_4, which roughly corresponds to the moment when the first legitimate fragment reassembles with the spoofed fragments in the client's fragment buffer. Now suppose the fragment buffer caches for 30 sec. This means that timestamps T_2 and T_3 (from attacker's second spoofed fragment) can sit in the fragment buffer for anywhere from 0 to 30 sec before reassembly at time

T_4 (Figure 6). Thus, the offset θ in the reassembled packet can vary in the range of 0 to 30 sec, causing jitter ψ to be about ≈ 30 sec and the attacker to fail `TEST11`.

Hurdle 2: IPID. Our attacker must ensure that the IPID of the spoofed fragments planted in the fragment buffer (the left two fragments in Figure 6) match the IPID of the legitimate fragments sent by the server (the right two fragments in Figure 6); this way, the spoofed fragments will properly reassemble with the legitimate fragments.

Surmounting these hurdles. To surmount the first hurdle, our attacker ensures that the client's fragment buffer always contains fresh copies of the second spoofed fragment that are no more than 1 sec old. Suppose that the client's fragment cache is a FIFO queue that holds a maximum of n fragments. (Windows XP has $n = 100$ [30], and the Linux kernel in [26] has $n = 64$) . Then, every second, the attacker sends the client $n/2$ copies of its first spoofed fragment (each with different IPIDs), and $n/2$ copies of the second spoofed fragment, per Figure 6. Each second spoofed fragment has (1) IPID corresponding to one of the first spoofed fragments, and (2) timestamps T_2 and T_3 corresponding to the (legitimate) time that the fragment was sent plus some constant value (*e.g.,* $x = +10$ mins, where x represents how far the attacker wants to shift the client forward/backward in time). Thus, every second, a fresh batch of n fragments evicts the old batch of n fragments. The reassembled packets have offset within ≈ 1 sec, so that jitter $\psi \approx 1$ sec, and the attacker passes `TEST11`.

To surmount the second hurdle, our attack exploits the fact that IPIDs are often predictable. Several policies for setting IPID exist in the wild, including: *globally-incrementing*, *i.e.,* the OS increments IPID by one for every sent packet, *per-destination-incrementing*, *i.e.,* the OS increments IPID by one every packet sent to a particular destination IP, and *random*, *i.e.,* the IPID is selected at random for every packet [20]. Random IPIDs thwart our attacks. However, when the server uses an incrementing IPID policy, the following techniques allow our attacker to plant several copies of the spoofed fragments with plausible IPIDs (*cf.,* [20], [26], [30]):

Globally incrementing IPIDs: Before sending the client the $n/2$ copies of the spoofed fragments, our attacker pings the server to learn its IPID i, and sets IPID of its spoofed packets accordingly (*i.e.,* to $i+1, i+2, ..., i+n/2$).

Per-destination incrementing IPIDs: Gilad and Hertzberg [20] [30] show how per-destination incrementing IPIDs can be inferred by a puppet (adversarial applet/script that runs in a sandbox) on the client or server's machine, while Knockell and Crandall [30] show how to do this without puppets. Thus, at the start of our attack, our attacker can use [20], [30]'s techniques to learn the initial IPID i, then uses i to set IPIDs on the $n/2$ copies of its pairs of spoofed fragments. The choice of IPIDs depends on the *polling interval*, *i.e.,* the frequency at which the client queries the server. NTP default poll values range from $\tau = 6$ ($2^6 = 64$ sec) to $\tau = 10$ (1024 seconds) [43]. If the attacker knew that the client was polling every 64 seconds, it could send $n/2$ copies of the spoofed fragments with IPID $i+1$,....,$i+n/2$, and then increment i every 64 seconds.

More commonly, however, the polling interval is unknown. To deal with this, the attacker can predict the IPID under the assumption that the client and server consistently used the minimum (*resp.,* maximum) polling interval, and ensures that all possible IPIDs in between are planted in the buffer. As an example, suppose that 2048 seconds (30 mins) have elapsed since the attacker learned that the IPID is i. At one extreme, the client and server could have consistently used the minimum default polling interval of $2^\tau = 64$ sec; thus, $i_{\max} = i+2048/64 = i+32$. At the other extreme, the client and server could have consistently used the maximum default polling interval of $2^\tau = 1024$ sec; then $i_{\min} = i+2048/1024 = i+2$. Then, the attacker must send pairs of spoofed fragments with IPIDs ranging from $i_{\min} = i+2$ to $i_{\max} = i+32$. This works as long as the fragment buffer can hold $i_{\max} - i_{\min} > 30 \cdot 2$ fragments (as in *e.g.,* Linux [26] and Windows XP [30]). When $2(i_{\min} - i_{\max})$ exceeds the size of the fragment buffer n, the attacker repeats [20], [30]'s techniques to learn IPID again.

Moreover, to avoid having to plant so many IPIDs in the fragment buffer, the attacker can try making the polling interval more predictable. Our experiments show that if a server becomes "unreachable" (*i.e.,* stops responding to queries) for a short period, and starts to respond with packets with *poll* field $\tau_p = 6$, the ntpd v4.2.6 client will speed up its polling interval to ≈ 64 sec. To simulate "unreachable" behavior from off-path, the attacker can plant fragments with incorrect UDP checksum (*e.g.,* planting just the second spoofed fragment, but not the first, per Figure 6). Then, after some time, the attack begins with *poll* set to $\tau_p = 6$ in the first spoofed fragment.

E. Conditions required for our attack.

In summary, our attack succeeds for a given victim NTP server and victim NTP client if the following hold:

1) the server accepts and acts on *ICMP fragmentation needed* packets for a 68-byte MTU, and
2) the server uses globally-incrementing or per-destination-incrementing IPID, and
3) the client reassembles overlapping IPv4 fragments as in Outcome A of Figure 7.

F. Proof-of-concept implementation of our attack.

We implemented a proof of concept of our attack on three lab machines. Our server had per-destination incrementing IPID.

Server. Our victim server ran ntpd v4.2.8p2 on Linux 3.13.0-24-generic kernel which uses a per-destination incrementing IPID. This Linux kernel has configuration parameter *min_pmtu* that determines the minimum MTU to which the OS is willing to fragment packets upon receipt of an *ICMP fragmentation needed* packet; we manually set *min_pmtu* to 68, so that the server would satisfy the first condition of our attack.[15]

Client. Choosing the right client was a challenge. It is extremely difficult to find documentation on overlapping fragment reassembly policies for popular OSes. Moreover, these policies change over time. For instance, in 2005 Novak [51] found that MAC OS reassembles as in Outcome A, but our

[15]The default value for *min_pmtu* in Linux exceeds 500 bytes [60], so that the vast majority of NTP servers running on Linux should not be vulnerable to our attack. (Linux 2.2.13 is one notable exception; see Section VI-G.) However, our measurements in Section VI-G indicate that servers in the wild do fragment to a 68-byte MTU; we just use Linux 3.13.0 as a proof-of-concept in our lab.

July 2015 experiments indicated that MAC OS X v10.9.5 reassembles as in Outcome B. After testing various OSes, we tried using the Snort IDS to emulate a common network topology (Section VI-H), where a middlebox reassembles fragmented packets before passing them on to end hosts [10]. We set up Snort in inline mode on a VM in front of another VM with the ntpd v4.2.6 client. Unfortunately, Snort's *frag3* engine, which reassembles overlapping IPv4 fragments according to various policies, exhibited buggy behavior with UDP (even though it worked fine with the ICMP fragments used in [51]). Finally, we gave up and wrote our own fragment buffer in python and scapy, running it on an Linux 3.16.0-23-generic OS with ntpd v4.2.6p5.

Our fragment buffer code had two parts. The first part uses scapy's sniff function to detect IPv4 fragments, and then sends them to our fragment buffer, which reassembles them and passes them back to the OS. The second part uses nfqueue to drop packets that were reassembled by the OS and pass packets reassembled by our fragment buffer. The fragment buffer itself is a FIFO queue with capacity $n = 28$ fragments and timeout $t = 30$ sec. Fragments older than t are evicted from the queue. When the queue is full, a newly-arrived fragment evicts the oldest fragment. The buffer reassembles packets according to the 'First' policy[16] in [63] (*i.e.,* Outcome A in Figure 6).

Attacker. Our attacker machine ran code written in scapy. Before the attack starts, we let the NTP client synchronize to the server. After that, our attacker machine should infer the IPID the server uses to send mode 4 packets to the client; rather than reimplementing the IPID-inference techniques of [20], [30], we just have the client machine send the initial IPID i directly to the attack machine. At this point, the client no longer reveals any more information to the attacker, and the attack starts. The attacker first sends the server a spoofed *ICMP fragmentation needed* packet requesting fragmentation to a 68-byte MTU for the client; the server caches the request for 10 minutes, and starts sending the two rightmost fragments in Figure 6. To keep the server fragmenting, the attacker send a fresh *ICMP fragmentation needed* every 10 minutes.

Per Section VI-D, each second our attacker needs to send the client a fresh batch of $n/2$ pairs of the two leftmost fragments in Figure 6. Each pair had IPID in $\{(i+1), ..., (i+n/2-1)\}$, with i incremented every 70 seconds.[17] Our attack machine was a fairly lame eight-year old Fujitsu x86_64 with 1.8GB of memory running Linux 3.16.0-24-generic, and thus could only manage to send thirteen pairs of the required fragments within 1 second. We therefore set the size of the FIFO queue on our client machine to $n = 28$. Our attacker uses the fragmentation attack to launch a "small-step-big-step" attack (Section IV): First, it sets receive and transmit timestamps T_2 and T_3 in its second spoofed fragment to shift the client 10 minutes back in time. Once the client enters 'STEP' mode, it sets T_2 and T_3 to shift the client one day back in time. (Note that an off-path attacker can check that the client is in 'STEP' mode by querying the client and checking for 'STEP' in the reference ID of the response [43, Fig. 13].)

[16]The 'First' policy of [63] requires reassembly to prefer the fragment that was received earliest by fragment buffer.

[17]NTP uses a randomized algorithm to set the polling interval. Our client had not been running for long, so its polling interval was $\tau = 6$ (64 sec), which translates to intervals randomly chosen from the discrete set $\{64, 65, 66, 67\}$ sec. We therefore increment i every 70 seconds. However, per Section VI-D, (1) an off-path attacker can push the polling interval down to $\tau = 6$ by using fragmentation to make the server to look like it has become 'unreachable', and (2) if the fragment buffer has large n, our attack can accommodate larger variation in the polling interval (*e.g.,* MAC OS X has $n = 1024$ [30]).

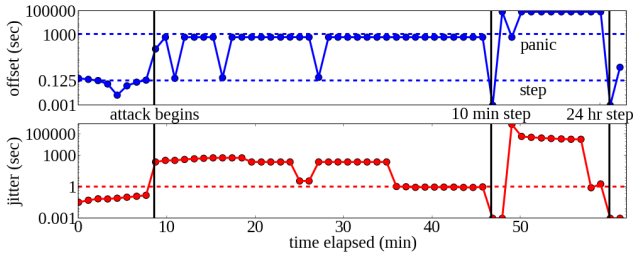

Fig. 8. Absolute value of offset θ (above) and jitter ψ (below) computed by the client during a proof-of-concept implementation of our attack.

Results (Figure 8). We plot the results of one run of our attack, obtained from the client's ntpq program. We plot offset θ (equation (2)) computed by the client for each mode 4 packet the client (thinks it) received from the server. The horizontal lines on the offset plot represent NTP's default panic threshold (1000 sec) and 'STEP' threshold (125 ms). We also plot jitter ψ (equation (3)) computed by the client from its eight most recent offset samples. Recall that the client will only synchronize to the server if $\psi < 1$ sec (Section V-E,VI-D). Before the attack begins, the client and server are synchronized and offset is small. Once the attack begins, offset jumps to about 600 seconds (10 minutes). Figure 8 also shows some spikes where the offset jumps back down to a few msec. These spikes occur during cache misses, when our attacker fails to plant fragments with the right IPID in the fragment buffer; this allow the two legitimate fragments to reassemble so that the client gets sample of the correct time. The attacker pays a price each time a cache miss causes an inconsistency in the offset values; for example, at time 25 mins, the attacker crafts enough packets to force the jitter to about 10 sec, but two samples later it suffers a cache miss, causing jitter to jump to about 200K sec. Eventually, the attacker crafts enough packets to keep jitter below 1 sec for some period of time, and the client accepts the time, enters 'STEP' mode, and clears its state. Once in 'STEP' mode, the attacker manages to craft nine consecutive packets, causing jitter to drop below 1 sec and sending the client back in time for another 24 hours.

G. Measuring the attack surface: Servers.

How often are the conditions required for our attack (Section VI-E) satisfied in the wild? We answer this question by scanning our dataset of 13M NTP servers (Section III-B) to find servers satisfying the two conditions for our attack per Section VI-E: (1) fragmenting to a 68-byte MTU, and (2) using incrementing IPID. To avoid harming live NTP servers with this scan, we send only ICMP packets or mode 3 NTP queries (which do not set time on the server).

Fragmenting to 68-byte MTU. To find NTP servers that fragment packets to a 68-byte MTU, we send each server in our list (1) an NTP mode 3 query and capture the corresponding NTP mode 4 response, and then (2) send an *ICMP fragmentation needed* packet requesting fragmentation to a 68-bytes for NTP packets sent to our measurement machine (as per the packet in Figure 5, where UDP checksum is zero and IPID inside the

IPID	Per-Dest	Globally incrementing					
behavior	$\Gamma = 1$	$\Gamma = 10$	$\Gamma = 25$	$\Gamma = 50$	$\Gamma = 100$	$\Gamma = 250$	$\Gamma = 500$
# servers	2,782	5,179	2,691	533	427	135	55

TABLE VI. IPID BEHAVIOR OF NON-BAD-TIMEKEEPERS SATISFYING CONDITIONS (1), (2) OF SECTION VI-E.

ICMP payload is that in the captured mode 4 NTP response), and finally (3) send another NTP mode 3 query. If the server fragments the final mode 4 NTP response it sends us, we conclude it satisfies the first condition for our attack.

Server IPID behavior. Next, we check the IPID behavior of each server that was willing to fragment to a 68-byte MTU. To do this, we send each IP five different NTP mode 3 queries, interleaving queries so that at about 30 sec elapse between each query to an IP. We then check the IPIDs for the mode 4 responses sent by each IP. If IPID incremented by one with each query, we conclude the server is vulnerable because it uses a per-destination-incrementing IPID. Otherwise, we determine the gap between subsequent IPIDs; if all gaps were less than a threshold Γ (for $\Gamma = \{10, 25, 100, 250, 500\}$), we conclude that the server uses a globally-incrementing IPID.

Results of server scan. Out of the 13M servers we scanned, about 24K servers were willing to fragment to a 68-byte MTU. 10K of these servers have bigger problems than just being vulnerable to our attacks: they were either unsynchronized (*i.e.,* either stratum 0 or stratum 16) or bad timekeepers (*i.e.,* with offset $\theta > 10$ sec). However, we did find 13,081 'functional' servers that fragment to a 68-byte MTU. As shown in Table VI, the vast majority (11,802 servers) of these are vulnerable to our attack because they use an incrementing IPIDs that grow slowly within a 30-second window. In fact, most use a globally-incrementing IPID, which is easier to attack than a per-destination IPID (see Section VI-D).

Who are these vulnerable servers? The majority 87% (10,292 servers) are at stratum 3, but we do see 14 vulnerable stratum 1 servers and 660 vulnerable servers with stratum 2. Indexing these with our (very incomplete) topology data, we find that 11 of these servers are at the root of subtrees with over 1000 clients, and 23 servers have over 100 clients. One vulnerable stratum 2 server, for example, is in South Korea and serves over 19K clients, another with over 10K clients is in a national provider in the UK, one with over 2K clients serves a research institute in Southern Europe, and two with over 7K clients are inside a Tier 1 ISP.

When we cross-reference these servers to our *rv* data from May 2015, we find that the vast majority (9,571 out of the 11,803 vulnerable servers) are running Linux 2.2.13; the other significant group is 1,295 servers running some version of "SunOS". We note that *not* every Linux 2.2.13 server in our *rv* dataset fragmented to a 68 byte MTU; 688 of the servers running on Linux 2.2.13 in our *rv* data responded to our final NTP query with an unfragmented NTP response, even though they had been sent a valid *ICMP fragmentation needed* packet, possibly because of middleboxes that drop ICMP packets.

H. Measuring the attack surface: Clients.

Determining how many clients in the wild satisfy the third condition of our attack per Section VI-E was a significantly more complex enterprise. To measure how an NTP client reassembles overlapping IPv4 fragments, we can use [51], [63]'s technique of sending fragmented ping packets. To check for reassembly per Outcome A in Figure 7, we send four ping fragments with offsets corresponding exactly to those in Figure 6. If the server reassembles them as in Outcome A, the reassembled ping packet will have a correct ICMP checksum and elicit a ping response from the server; otherwise, the server will ignore them. Figure 9 shows the four ping fragments and how they would be reassembled per Outcome A. We repeat this with four other ping fragments to check for Outcome B.

Before we could deploy this technique in the wild, we hit an important complication: Teardrop [9], an ancient implementation bug (from 1997) with devastating consequences. In a teardrop attack, the attacker sends two overlapping IPv4 fragments to an OS, and the OS crashes. Most OSes were patched for this over fifteen years ago, but some firewalls and middleboxes still alarm or drop packets when they see overlapping IPv4 fragments. Worse yet, legacy operating systems may not be patched, and new IP stacks might have reintroduced the bug. This is a big problem for us: this measurement technique inherently requires overlapping IPv4 fragments, and thus inherently contains a teardrop attack. We therefore cannot run this measurement on all 13M NTP servers we found in the wild, since we don't know what OSes they might be running. Instead, we deal with this in two ways.

First, we have developed a website that allows operators to check if their own NTP clients could be vulnerable to our attack because they reassemble packets as in Outcome A. (https://www.cs.bu.edu/~goldbe/NTPattack.html) To prevent the site itself from becoming a teardrop attack vector, we require users running the measurement to be on the same IP prefix as the measured NTP client.

Second, we can send our measurements to NTP servers that we know are patched for Teardrop. Teardrop affects version of Linux previous to 2.0.32 and 2.1.63 [9]; thus, we can use the *rv* data from the openNTPproject to determine which servers are running patched Linux versions, and send our measurements to those servers only. We did this for 384 servers that responded to *rv* queries with "Linux/3.8.13", a kernel released in May 2013, well after Teardrop was patched. Five servers responded with pings reconstructed as in Outcome A, 51 servers with pings reconstructed as in Outcome B.

This is interesting for two reasons. Most obviously, this gives evidence for the presence of reassembly per Outcome A in the wild, which means there are NTP clients that are vulnerable to our attack. But it also suggests that this fragmentation reassembly is not always done by the endhost itself; if it had, all 384 servers would have responded in the same way. Thus, we speculate that these five servers responded to our ping because they were sitting behind a middlebox that performs fragment reassembly for them [10].

I. Recommendations: Fragmentation still considered harmful.

Our measurements suggest that the attack surface for our NTP fragmentation attack is small but non-negligible. Thousands of NTP servers satisfy the conditions required by our attack (Section VI-E). However, our attack only succeeds if the victim client is synchronized to a vulnerable server, and reassembles fragmented packets according the third condition required for our attack (Section VI-E). Unfortunately, we could

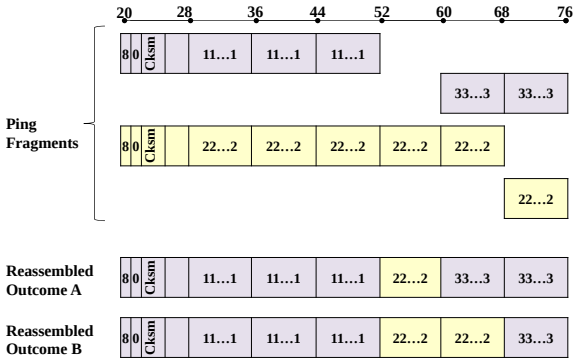

Fig. 9. Ping packets for measuring fragmentation reassembly policies.

not safely measure which NTP clients reassemble packets in this way, although we do find evidence of vulnerable clients.

Perhaps the simplest way to protect the NTP ecosystem is to ensure that any OS running an NTP server does not fragment packets to a 68 byte MTU; indeed, many OSes already do this (*e.g.,* Linux [60], Windows [41]). On the client side, the OS should drop overlapping NTP fragments, as should middleboxes that reassemble IPv4 fragments like [10].

One might naturally wonder whether our attack is thwarted by UDP source-port randomization. (That is, what if the client chose a random 16-bit value for the UDP source port [35], instead of always setting it to 123?) Unfortunately, the attacker may still be able to use IPv4 fragmentation to circumvent this defense, even without knowing the source port. Suppose conditions (1), and (2) of Section VI-E hold, and consider modifying our attack as follows: replace the first spoofed fragment on the left of Figure 6 with an identical fragment that has its UDP source port and destination port fields sliced off. (The replacement fragment has fragment offset of 24 bytes.) Then, if the client reassembles the packet according to the 'First' policy of [63], then the packet will look just like the one reassembled per Outcome A in Figure 7, except with the legitimate UDP source/dest ports, and the attack will succeed. Thus, while UDP source-port randomization raises the bar for our attack, we do not consider it to be a sufficient defense.

VII. Related work

NTP security. While NTP-based DDoS amplification attacks have generated widespread interest (see *e.g.,* [14]), there is less research [13], [29], [44], [46], [61] on the implications of shifting time via NTP attacks. A few researchers [44], [46], [61], [62] have considered the implications of attacks on NTP traffic, and [61], [62] demonstrated on-path attacks on timing clients that update their clocks at predictable intervals. We consider on-path attacks on the full NTP implementation (ntpd), which does *not* perform clock updates in a predictable fashion, and present new off-path attacks. Complementary to our work are efforts to identify software bugs in ntpd [50], [58], including several defects that were made public at the same time as our work [11]; because ntpd typically operates as root on the host machine, we expect that interest in this area will only continue to increase. Finally, our work is also related to older NTP measurement studies [45], [48], as well as the recent work of [14]; while [14] looked at DDoS attacks, we focus on the integrity of timing information and vulnerabilities identified by our attacks. Concurrent to our work is an interesting new study on using NTP to measure network latency [17].

IPv4 Fragmentation. Our work is also related to research on exploiting IPv4 packet fragmentation for *e.g.,* off-path attacks on operating systems [9], DNS resolvers [26], TCP connections [20], [22], and to evade intrusion detection systems [6], [51], [56], [63] and exploit side channels [30]. Unlike most prior work, however, we had to use fragmentation to craft a *stream* of self-consistent packets, rather than a single packet. Our attack also exploits problems with overlapping IPv4 fragments [6], [51], [63] and tiny IPv4 fragments, and should provide some extra motivation for OSes/middleboxes to drop tiny/overlapping fragments, rather than reassemble them.

VIII. Conclusion

Our results suggest four ways to harden NTP:

1) In Section IV we discuss why freshly-restarted ntpd clients running with the `-g` configuration (which is the default installation for many OSes) are vulnerable to quick time shifts of months or years. We also present a 'small-step-big-step' attack, captured in CVE-2015-5300, that allows an on-path attacker to stealthily shift a time on a freshly-restarted ntpd client. Different versions of the small-step-big-step attack succeed on ntpd v4.2.6 and ntpd v4.2.8. To protect against these attacks, users can either stop using the `-g` configuration, or monitor their systems for suspicious reboots of the OS or of ntpd. Section IV-C also has recommendations for implementors that wish to patch against our small-step-big-step attack.

2) We showed how NTP's rate-limiting mechanism, the KoD, can be exploited for off-path denial-of-service attacks on NTP clients (Section V-C). ntpd 4.2.8p3, the current reference implementation, is vulnerable to an extremely low-rate version of this attack (requiring about 1 packet per pre-configured server) by spoofing KoD packets. This attack has been captured in CVE-2015-7704 and patched in ntpd 4.2.8p4. Moreover, even if this vulnerability (that allows KoD packets to be spoofed from off-path), we show that an off-path attacker can still to launch denial-of-service attacks using a 'priming the pump' technique (Section V-B). This attack has been captured in CVE-2015-7705. As we argue in Section V-G, we believe that NTP should either (1) eliminate its KoD functionality, (2) require NTP clients to cryptographically authenticate their queries to NTP servers, or (3) adopt more robust rate limiting techniques, like [68].

3) In Section VI we showed how off-path attacker can use IPv4 fragmentation to hijack an NTP connection from client to server. Because our attack requires server and client to run on operating systems that use less-common IPv4 fragmentation policies, we have used a measurement study to quantify its attack surface, and found it to be small but non-negligible. As we argue in Section VI-I, NTP servers should run on OSes that use a default minimum MTU of ≈ 500 bytes, as in recent versions of Linux and Windows [41], [60]. OSes and middleboxes should also drop overlapping IPv4 fragments. We have also set up a website where operators can test their NTP clients for vulnerability to our fragmentation attacks.[18]

[18] https://www.cs.bu.edu/~goldbe/NTPattack.html

4) Each of our attacks has leveraged useful information that clients leak in the *reference ID* field of their mode 4 response packets (Figure 1). Moreover, clients typically send mode 4 responses in response to any mode 3 query sent by any IP in the Internet. (In Section IV, we use the fact that reference ID leaks that the client was in the 'INIT' or 'STEP' state. In Section V-C our off-path attacker used the reference ID field to learn the IPs of client's servers.) Thus, it would be worthwhile to consider limiting the information leaked in the reference ID field. In fact, RFC 5905 [43, pg 22] already requires IPv6 addresses to be hashed and truncated to 32 bits before being recorded as a reference ID. Of course, this approach is vulnerable to trivial dictionary attacks (with a small dictionary, namely, the IPv4 address space).

There are more robust ways to obfuscate the reference ID. Indeed, the primary purpose of the reference ID is to prevent timing loops, where client A takes time from client B who takes time from client A [66]. One idea is to use a salted hash,, in an approach analogous to password hashing. Upon sending a packet, client A chooses a a fresh random number as the 'salt', includes the salt in the NTP packet (perhaps as the lower-order bits of the reference timestamp (Figure 1)), and sets the reference ID in the packet to Hash(IP, salt), where IP is the IP address of the server B from which A takes time. B can check for a timing loop by extracting the salt from the packet, taking B's own IP, and checking that Hash(IP, salt) matches the value in the packet's reference ID; if yes, there is a timing loop, if no, there is not. This approach requires no state at A or B, and de-obfuscating the IP requires an attacker to recompute the dictionary for each fresh salt. This approach, however, comes with the caveat that an committed attacker could still launch dictionary attacks on the salted hash.

Our work may also motivate the community to take another look at cryptographic authentication for NTP [23], [46], [57].

ACKNOWLEDGEMENTS

We thank the Network Time Foundation, NTPsec, the RedHat security team and Cisco for promptly issuing patches and working with vendors to mitigate our attacks.

We are especially grateful to Jared Mauch for giving us access to the openNTPproject's data and servers and for helping us coordinate the disclosure of this work, and to Miroslav Lichvar for his careful review of our work, especially the small-step-big-step attack. We thank Quinn Shamblin, Lora Fulton, Paul Stauffer and Joe Szep for IT and research support, Haya Shulman and Yossi Gilad for discussions about packet fragmentation, Nadia Heninger for discussions about TLS, Dimitris Papadopoulos for discussions about DNS, Ethan Heilman for discussions about the RPKI, and Leonid Reyzin for general wisdom. We had productive discussions about NTP security with Majdi Abbas, Daniel Franke, Sue Graves, Matt Van Gundy, Graham Holmes, David Mills, Danny Mayer, Hai Li, Karen ODonoghue, Clay Seaman-Kossmeyer, Harlan Stenn, Eric S. Raymond, Amar Takar, Dave Ward, and Florian Weimer. This research was supported, in part, by NSF awards 1347525, 1350733 and 1012910 and a gift from Cisco.

REFERENCES

[1] Autokey Configuration for NTP stable releases. The NTP Public Services Project: http://support.ntp.org/bin/view/Support/ConfiguringAutokey (Accessed: July 2015).

[2] DoD Customers : Authenticated NTP. http://www.usno.navy.mil/USNO/time/ntp/dod-customers (Accessed: July 2015).

[3] The NIST authenticated ntp service. http://www.nist.gov/pml/div688/grp40/auth-ntp.cfm (Accessed: July 2015), 2010.

[4] Amazon. Simple storage service (s3): Signing and authenticating rest requests. http://docs.aws.amazon.com/AmazonS3/latest/dev/RESTAuthentication.html (Accessed Aug 2015).

[5] Axel K. ntpd access restrictions, section 6.5.1.1.3. allow queries? http://support.ntp.org/bin/view/Support/AccessRestrictions, February 16 2015.

[6] M. Baggett. Ip fragment reassembly with scapy. *SANS Institute InfoSec Reading Room*, 2012.

[7] E. Barker and A. Roginsky. Transitions: Recommendation for transitioning the use of cryptographic algorithms and key lengths. *NIST Special Publication. http://csrc.nist.gov/publications/nistpubs/800-131A/sp800-131A.pdf*, 800:131A, 2011.

[8] L. Bicknell. NTP issues today. Outages Mailing List http://mailman.nanog.org/pipermail/nanog/2012-November/053449.html, November 2012.

[9] BUGTRAQ mailing list. Linux and windows ip fragmentation (teadrop) bug, 1997. http://insecure.org/sploits/linux.fragmentation.teardrop.html.

[10] Checkpoint. Ip fragments (UTM-1 appliance). https://www.checkpoint.com/smb/help/utm1/8.1/2032.htm.

[11] A. Chiu. Cisco identifies multiple vulnerabilities in network time protocol daemon (ntpd). TALOS Blog http://blog.talosintel.com/2015/10/ntpd-vulnerabilities.html, October 2015.

[12] D. Cooper, E. Heilman, K. Brogle, L. Reyzin, and S. Goldberg. On the risk of misbehaving RPKI authorities. *HotNets XII*, 2013.

[13] corbixgwelt. Timejacking & bitcoin: The global time agreement puzzle (culubas blog), 2011. http://culubas.blogspot.com/2011/05/timejacking-bitcoin_802.html (Accessed Aug 2015).

[14] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir. Taming the 800 pound gorilla: The rise and decline of ntp ddos attacks. In *Proceedings of the 2014 Internet Measurement Conference*, pages 435–448. ACM, 2014.

[15] M. d'Itri. Hacking team and a case of bgp hijacking. http://blog.bofh.it/id_456, July 2015.

[16] DropBox. Core API. https://www.dropbox.com/developers/core/docs (Accessed Aug 2015).

[17] R. Durairajan, S. K. Mani, J. Sommers, and P. Barford. Time's forgotten: Using ntp to understand internet latency. *HotNets'15*, November 2015.

[18] Z. Durumeric, E. Wustrow, and J. A. Halderman. Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security*, pages 605–620. Citeseer, 2013.

[19] P. Eckersley and J. Burns. An observatory for the ssliverse. DEFCON'18, July 2010.

[20] Y. Gilad and A. Herzberg. Fragmentation considered vulnerable. *ACM Trans. Inf. Syst. Secur*, 15(4), 2013.

[21] S. Goldberg. Why is it taking so long to secure internet routing? *Communications of the ACM: ACM Queue*, 57(10):56–63, 2014.

[22] F. Gont. *RFC 5927: ICMP attacks on TCP*. Internet Engineering Task Force (IETF), 2010. https://tools.ietf.org/html/rfc5927.

[23] B. Haberman and D. Mills. *RFC 5906: Network Time Protocol Version 4: Autokey Specification*. Internet Engineering Task Force (IETF), 2010. https://tools.ietf.org/html/rfc5906.

[24] E. Hammer-Lahav. *RFC 5849: The OAuth 1.0 Protocol*. Internet Engineering Task Force (IETF), 2010. https://tools.ietf.org/html/rfc5849.

[25] E. Heilman, D. Cooper, L. Reyzin, and S. Goldberg. From the consent of the routed: Improving the transparency of the RPKI. *ACM SIGCOMM'14*, 2014.

[26] A. Herzberg and H. Shulman. Fragmentation considered poisonous, or: One-domain-to-rule-them-all. org. In *Communications and Network Security (CNS), 2013 IEEE Conference on*, pages 224–232. IEEE, 2013.

[27] D. Kaminsky. Black ops 2008: It's the end of the cache as we know it. *Black Hat USA*, 2008.

[28] K. Kiyawat. Do web browsers obey best practices when validating digital certificates? Master's thesis, Northeastern University, December 2014.

[29] J. Klein. Becoming a time lord - implications of attacking time sources. Shmoocon Firetalks 2013: https://youtu.be/XogpQ-iA6Lw, 2013.

[30] J. Knockel and J. R. Crandall. Counting packets sent between arbitrary internet hosts. In *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI'14)*, 2014.

[31] B. Knowles. NTP support web: Section 5.3.3. upstream time server quantity, 2004. http://support.ntp.org/bin/view/Support/SelectingOffsiteNTPServers#Section_5.3.3. (Accessed July 2015).

[32] J. Kohl and C. Neuman. *RFC 1510: The Kerberos Network Authentication Service (V5)*. Internet Engineering Task Force (IETF), 1993. https://tools.ietf.org/html/rfc1510.

[33] O. Kolkman, W. Mekking, and R. Gieben. *RFC 6781: DNSSEC Operational Practices, Version 2*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc6781.

[34] A. Langley. Revocation still doesn't work. https://www.imperialviolet.org/2011/03/18/revocation.html, April 29 2014.

[35] M. Larsen and F. Gont. *RFC 6056: Recommendations for Transport-Protocol Port Randomization*. Internet Engineering Task Force (IETF), 2011. https://tools.ietf.org/html/rfc6056.

[36] M. Lepinski and S. Kent. *RFC 6480: An Infrastructure to Support Secure Internet Routing*. Internet Engineering Task Force (IETF), 2012. https://tools.ietf.org/html/rfc6480.

[37] L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, and R. Wheeler. *RFC 2516: A Method for Transmitting PPP Over Ethernet (PPPoE)*. Internet Engineering Task Force (IETF), 1999. https://tools.ietf.org/html/rfc2516.

[38] K. A. Marzullo. *Maintaining the Time in a Distributed System*. PhD thesis, Stanford, 1984.

[39] J. Mauch. openntpproject: NTP Scanning Project. http://openntpproject.org/.

[40] D. Menscher. NTP issues today. Outages Mailing List http://mailman.nanog.org/pipermail/nanog/2012-November/053494.html, November 2012.

[41] Microsoft. MS05-019: Vulnerabilities in TCP/IP could allow remote code execution and denial of service, 2010. https://support.microsoft.com/en-us/kb/893066 (Accessed July 2015).

[42] I. Miller. *RFC 3128 (Informational): Protection Against a Variant of the Tiny Fragment Attack*. Internet Engineering Task Force (IETF), 2001. https://tools.ietf.org/html/rfc3128.

[43] D. Mills, J. Martin, J. Burbank, and W. Kasch. *RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification*. Internet Engineering Task Force (IETF), 2010. http://tools.ietf.org/html/rfc5905.

[44] D. L. Mills. *Computer Network Time Synchronization*. CRC Press, 2nd edition, 2011.

[45] N. Minar. A survey of the ntp network, 1999.

[46] T. Mizrahi. *RFC 7384 (Informational): Security Requirements of Time Protocols in Packet Switched Networks*. Internet Engineering Task Force (IETF), 2012. http://tools.ietf.org/html/rfc7384.

[47] M. Morowczynski. Did your active directory domain time just jump to the year 2000? Microsoft Server & Tools Blogs http://blogs.technet.com/b/askpfeplat/archive/2012/11/19/did-your-active-directory-domain-time-just-jump-to-the-year-2000.aspx, November 2012.

[48] C. D. Murta, P. R. Torres Jr, and P. Mohapatra. Characterizing quality of time and topology in a time synchronization network. In *GLOBECOM*, 2006.

[49] P. Mutton. Certificate revocation: Why browsers remain affected by heartbleed. http://news.netcraft.com/archives/2014/04/24/certificate-revocation-why-browsers-remain-affected-by-heartbleed.html, April 24 2014.

[50] National Vulnerability Database. CVE-2014-9295: Multiple stack-based buffer overflows in ntpd in ntp before 4.2.8, 2014. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9295.

[51] J. Novak. Target-based fragmentation reassembly. Technical report, Sourcefire, Incorporated, 2005.

[52] A. Peterson. Researchers say u.s. internet traffic was re-routed through belarus. that's a problem. *Washington Post Blogs: The Switch*, November 20 2013.

[53] J. Postel. *RFC 768: User Datagram Protocol*. Internet Engineering Task Force (IETF), 1980. https://tools.ietf.org/html/rfc768.

[54] J. Postel. *RFC 791: INTERNET PROTOCOL: DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION*. Internet Engineering Task Force (IETF), 1981. https://tools.ietf.org/html/rfc791.

[55] J. Postel. *RFC 792: INTERNET CONTROL MESSAGE PROTOCOL*. Internet Engineering Task Force (IETF), 1981. https://tools.ietf.org/html/rfc792.

[56] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., 1998.

[57] S. Röttger. Analysis of the ntp autokey procedures. Master's thesis, Technische Universitt Braunschweig, 2012.

[58] S. Rottger. Finding and exploiting ntpd vulnerabilities, 2015. http://googleprojectzero.blogspot.co.uk/2015/01/finding-and-exploiting-ntpd.html.

[59] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. *RFC 6960: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol OCSP*. Internet Engineering Task Force (IETF), 2013. https://tools.ietf.org/html/rfc6960.

[60] C. Schramm. Why does linux enforce a minimum MTU of 552?, 2012. http://cschramm.blogspot.com/2012/12/why-does-linux-enforce-minimum-mtu-of.html.

[61] J. Selvi. Bypassing http strict transport security. *Black Hat Europe*, 2014.

[62] J. Selvi. Breaking ssl using time synchronisation attacks. *DEFCON'23*, 2015.

[63] U. Shankar and V. Paxson. Active mapping: Resisting nids evasion without altering traffic. In *Symposium on Security and Privacy*, pages 44–61. IEEE, 2003.

[64] D. Sibold, S. Roettger, and K. Teichel. *draft-ietf-ntp-network-time-security-10: Network Time Security*. Internet Engineering Task Force (IETF), 2015. https://tools.ietf.org/html/draft-ietf-ntp-network-time-security-10.

[65] H. Stenn. Antw: Re: Proposed REFID changes. NTP Working Group Mailing List http://lists.ntp.org/pipermail/ntpwg/2015-July/002291.html, July 2015.

[66] H. Stenn. NTP's REFID. http://nwtime.org/ntps-refid/, September 2015.

[67] H. Stenn. Securing the network time protocol. *Communications of the ACM: ACM Queue*, 13(1), 2015.

[68] P. Vixie. Rate-limiting state. *Communications of the ACM: ACM Queue*, 12(2):10, 2014.

[69] P. Vixie and V. Schryver. Dns response rate limiting (dns rrl). http://ss.vix.su/~vixie/isc-tn-2012-1.txt, April 2012.

[70] VMware. Timekeeping in vmware virtual machines: vsphere 5.0, workstation 8.0, fusion 4.0, 2011. http://www.vmware.com/files/pdf/Timekeeping-In-VirtualMachines.pdf (Accessed July 2015).

[71] L. Zhang, D. Choffnes, D. Levin, T. Dumitras, A. Mislove, A. Schulman, and C. Wilson. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. In *Internet Measurement Conference (IMC'14)*, pages 489–502, 2014.