The Security of NTP's Datagram Protocol

Aanchal Malhotra*, Matthew Van Gundy†, Mayank Varia*,
Haydn Kennedy*, Jonathan Gardner†, Sharon Goldberg*

*Boston University
†Cisco (ASIG)

Abstract. For decades, the Network Time Protocol (NTP) has been used to synchronize computer clocks over untrusted network paths. This work takes a new look at the security of NTP's datagram protocol. We argue that NTP's datagram protocol in RFC5905 is both underspecified and flawed. The NTP specifications do not sufficiently respect (1) the conflicting security requirements of different NTP modes, and (2) the mechanism NTP uses to prevent off-path attacks. A further problem is that (3) NTP's control-query interface reveals sensitive information that can be exploited in off-path attacks. We exploit these problems in several attacks that remote attackers can use to maliciously alter a target's time. We use network scans to find millions of IPs that are vulnerable to our attacks. Finally, we move beyond identifying attacks by developing a cryptographic model and using it to prove the security of a new backwards-compatible client/server protocol for NTP.

1 Introduction

Millions of hosts [11,21,24,29,33] use the Network Time Protocol (NTP) [27] to synchronize their computer clocks to public Internet timeservers (using NTP's client/server mode), or to neighboring peers (using NTP's symmetric mode). Over the last few years, the security of NTP has come under new scrutiny. Along with significant attention paid to NTP's role in UDP amplification attacks [11,20], there is also a new focus on attacks on the NTP protocol itself, both in order to maliciously alter a target's time (*timeshifting attacks*) or to prevent a target from synchronizing its clock (*denial of service (DoS) attacks*) [21,41]. These attacks matter because the correctness of time underpins many other basic protocols and services. For instance, cryptographic protocols use timestamps to prevent replay attacks and limit the use of stale or compromised cryptographic material (*e.g.,* TLS [19,36], HSTS [35], DNSSEC, RPKI [21], bitcoin [10], authentication protocols [19,21]), while accurate time synchronization is a basic requirement for various distributed protocols.

1.1 Problems with the NTP specification.

We start by identifying three fundamental problems with the NTP specification in RFC5905, and then exploit these problems in four different off-path attacks on *ntpd*, the "reference implementation" of NTP.

Problem 1: Lack of respect for basic protection measures. The first issue stems from a lack of respect for `TEST2`, the mechanism that NTP uses to prevent off-path attacks. Off-path attacks are essentially the weakest (and therefore the most scary) threat model that one could consider for a networking protocol. An *off-path attacker* cannot eavesdrop on the NTP traffic of their targets, but can *spoof* IP packets *i.e.,* send packets with a bogus source IP. This threat model captures 'remote attacks' launched by arbitrary IPs that do not occupy a privileged position on the communication path between the parties. (See Figure 2.)

NTP attempts to prevent off-path attacks much in the same way that TCP and UDP do: every client query includes a nonce, and this nonce is reflected back to the client in the server's response. The client then checks for matching nonces in the query and response, *i.e.,* "`TEST2`". Because an off-path attacker cannot see the nonce (because it cannot eavesdrop on traffic), it cannot spoof a valid server response. Despite the apparent simplicity of this mechanism, its specification in RFC5905 is flawed and leads to several off-path attacks.

Problem 2: Same code for different modes. NTP operates in several different modes. Apart from the popular *client/server mode* (where the client synchronizes to a time server), NTP also has a *symmetric mode* (where neighboring peers take time from each other), and several other modes. RFC5905 recommends that all of NTP's different modes be processed by the same codepath. However, we find that the security requirements of client/server mode and symmetric mode conflict with each other, and result in some of our off-path attacks.

Problem 3: Leaky control queries. NTP's control-query interface is not specified in RFC5905, but its specification does appear in the obsoleted RFC1305 [25] from 1992 and a new IETF Internet draft [26]. We find that it can be exploited remotely to leak information about NTPs internal timing state variables. While the DDoS amplification potential of NTP's control query interface is well known [11, 20], here we show that it is also a risk to the correctness of time.

We exploit these three problems to find working off-path attacks on ntpd (Section 3-4, Appendix A), and use IPv4 Internet scans to identify millions of IPs that are vulnerable to our attacks (Section 5). The first three attacks maliciously shift time on a client using NTP's client/server mode, and the fourth prevents time synchronization in symmetric mode.

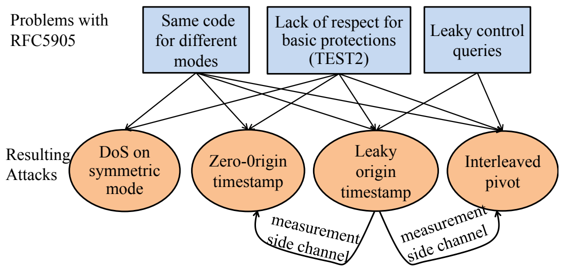

Fig. 1. Paper overview.

Attack 1: Leaky Origin Timestamp Attack (Section 4). Our network scans find a staggering 3.8 million IPs that leak the nonce used in `TEST2` in response to control queries made from arbitrary IPs (CVE-2015-8139). An off-path attacker can maliciously shift time on a client by continuously querying for this nonce, and using it to spoof packets that pass `TEST2`.

Attack 2: Zero-0rigin Timestamp Attack (Section 3.3 and Appendix A.1). This attack (CVE-2015-8138) follows from RFC5905, and is among the strongest timeshifting attacks on NTP that has been identified thus far. The attacker bypasses `TEST2` by spoofing server response packets with their nonce set to zero. We use leaky NTP control queries as a side-channel to measure the prevalence of this attack. We find 1.3 million affected IPs. However, we expect that the true attack surface is even larger, since this attack itself does not require the control-query interface, works on clients operating in default mode, and has been part of ntpd for seven years (since ntpd v4.2.6, December 2009).

Attack 3: Interleaved-Pivot Attack (Section 4). Our third off-path timeshifting attack (CVE-2016-1548) exploits the fact that NTP's client/server mode shares the same codepath as NTP's interleaved mode. First, the attacker spoofs a single packet that tricks the target into thinking that he is in interleaved mode. The target then rejects all subsequent legitimate client/server mode packets. This is a DoS attack (Section 4, Appendix A.2). We further leverage NTP's leaky control queries to convert this DoS attack to an off-path timeshifting attack. NTP's control-query interface also leaks the nonce used in the special version of `TEST2` used in interleaved mode. The attacker spoofs a sequence of interleaved-mode packets, with nonce value revealed by these queries, that maliciously shifts time on the client. Our scans find 1.3 million affected IPs.

Attack 4: Attacks on symmetric mode (Appendix B). We then present security analysis of NTP's symmetric mode, as specified in RFC5905, and present off-path attacks that prevent time synchronization. We discuss why the security requirements of symmetric mode are at odds with that of client/server mode, and may have been the root cause of the zero-0rigin timestamp attack.

Disclosure. Our disclosure timeline is in Appendix E. Our research was done against ntpd v4.2.8p6, the latest version as of April 25, 2016. Since then, three versions have been released: ntpd v4.2.8p7 (April 26, 2016), ntpd v4.2.8p8 (June 2, 2016), ntpd v4.2.8p9 (November 21, 2016). Most of our attacks have been patched in these releases. We provide recommendations for securing the client/server mode in Section 7 and symmetric mode in Appendix B.4.

1.2 Provably secure protocol design.

Our final contribution is to go beyond attacks and patches, and identify a more robust security solution (Section 6) We propose a new backwards-compatible protocol for client/server mode that preserves the semantics of the timestamps in NTP packets (Figures 6, 7). We then leverage ideas from the universal composability framework [8] to develop a cryptographic model for attacks on NTP's datagram protocol. We use this model to prove (Section 6.3,6.4) that our protocol *correctly synchronizes time* in the face of both (1) *off-path attackers* when NTP is unauthenticated and (2) *on-path attackers* when NTP packets are authenticated with a MAC. We also use our model to prove similar results about a different protocol that is used by chronyd [2] and openntpd [4] (two alternate implementations of NTP). The chronyd/openntpd protocol is secure, but unlike our protocol, does not preserve the semantics of packet timestamps.

Our cryptographic model models both on-path attackers and off-path attackers. An on-path attacker can eavesdrop, inject, spoof, and replay packets, but *cannot* drop, delay, or tamper with legitimate traffic. An on-path attacker eavesdrops on a copy of the target's traffic, so it need not disrupt live network traffic, or even operate at line rate. For this reason, on-path attacks are commonly seen in the wild, disrupting TCP [43], DNS [13], BitTorrent [43], or censoring web content [9]. Meanwhile, we cannot prove that NTP provides correct time synchronization in the face of the traditional Man-in-The-Middle (MiTM) attacks (*aka.* 'in-path attacks') because an MiTM can always prevent time synchronization by dropping packets. Moreover, an MiTM can also bias time synchronization by delaying packets [30, 31].[1]

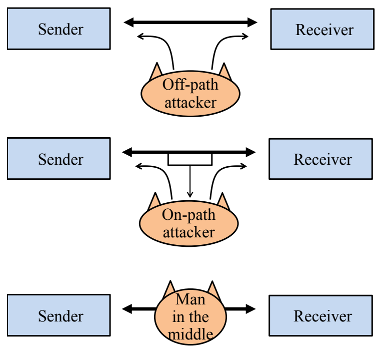

Fig. 2. Threat models.

Taking a step back, our work can be seen as a case study of the security risks that arise when network protocols are underspecified. It also highlights the importance of handling diverse protocol requirements in separate and rigourously tested codepaths. Finally, our network protocol analysis introduces new ways of reasoning about network attacks on time synchronization protocols.

1.3 Related work

Secure protocols. Our design and analysis of secure client/server protocols complement recent efforts to cryptographically secure NTP and its "cousin" PTP (Precision Time Protocol) [31]. Our interest is in securing the core datagram protocol used by NTP, which was last described in David Mill's book [28]. To the best of our knowledge, the security of the core NTP datagram protocol has never previously been analyzed. Meanwhile, our analysis assumes that parties correctly distribute cryptographic keys and use a secure MAC. A complementary stream of works propose protocols for distributing keys and performing the MAC, beginning with the Autokey protocol in RFC5906 [16], which was broken by Rottger [34], which was followed by NTS [38], ANTP [12], other works including [17, 32], and on-going activity in the IETF [1].

[1] This follows because time-synchronization protocols use information about the delay on the network path in order to accurately synchronize clocks (Section 2). A client cannot distinguish the delay on the forward path (from client to server) from the delay on the reverse path (from server to client). As such, the client simply takes the total round trip time δ (forward path + reverse path), and assumes that delays on each path are symmetric. The MiTM can exploit this by making delays asymmetric (*e.g.,* causing the delay on the forward path to be much longer than delay on the reverse path), thus biasing time synchronization.

Attacks. Our analysis of the NTP specification is motivated, in part, by discovery of over 30 ntpd CVEs between June 2015 to July 2016 [41]. These implementation flaws allow remote code execution, DoS attacks, and timeshifting attacks. Earlier, Selvi [35, 36] demonstrated MiTM timeshifting attacks on 'simple NTP (SNTP)' (rather than full-fledged NTP). Even earlier, work [10,19,28] considered the impact of timeshifting on the correctness of other protocols. The recent academic work [21] also attacks NTP, but our attacks are stronger. [21] presented attacks that are on-path (weaker than our off-path attacks), or off-path DoS attacks (weaker than our timeshifting attacks), or off-path time-shifting attacks that needed special client/server configurations (our Zero-0rigin Timestamp attack works in default mode). Also, our measurements find millions of vulnerable clients, while [21] finds thousands. Finally, NTP's broadcast mode is outside our scope; see [22, 31, 39] instead.

Measurement. Our work is also related to studies measuring the NTP ecosystem (in past decades) [29, 33], the use of NTP for DDoS amplification attacks [11], the performance of NIST's timeservers [37], and network latency [14]. Our attack surface measurements are in the same spirit as those in [21, 22], but we use a new set of NTP control queries. We also provide updated measurements on the presence of cryptographically-authenticated NTP associations.

2 NTP Background

NTP's default mode of operation is a hierarchical *client/server* mode. In this mode, timing queries are solicited by clients from a set of servers; this set of servers is typically static and configured manually. *Stratum i* systems act as servers that provide time to stratum $i + 1$ systems, for $i = 1, ...15$. Stratum 1 servers are at the root of the NTP hierarchy. Stratum 0 and stratum 16 indicate failure to synchronize. Client/server packets are not authenticated by default, but a Message Authentication Code (MAC) can optionally be appended to the packet. NTP operates in several additional modes. In *broadcast mode*, a set of clients listen to a server that broadcasts timing information. In *symmetric mode*, peers exchange timing information (Appendix B). There is also an *interleaved mode* for more accurate timestamping (Appendix A.2).

Fig. 3. Timestamps induced by the server response packet (mode 4).

T_1: *Origin timestamp.*	Client's local time when sending query.
T_2: *Receive timestamp.*	Server's local time when receiving query.
T_3: *Transmit timestamp.*	Server's local time when sending response.
T_4: *Destination timestamp.*	Client's local time when receiving response.

NTP's client/server protocol consists of a periodic two-message *exchange*. The client sends the server a query (*mode 3*), and the server sends back a response (*mode 4*). Each exchange provides a *timing sample*, which uses the four timestamps in Figure 3. All four timestamps are 64 bits long, where the first 32 bits are seconds elapsed since January 1, 1970, and the last 32 bits are fractional seconds. T_1, T_2, and T_3 are fields in the server response packet (mode 4) shown in Figure 4. The delay δ is an important NTP parameter [27] that measures the round trip time between the client and the server:

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

If there are symmetric delays on the forward and reverse network paths, then the difference between the server and client clock is $T_2 - (T_1 + \frac{\delta}{2})$ for the client query, and $T_3 - (T_4 - \frac{\delta}{2})$ for the server response. Averaging, we get *offset* θ:

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

A client does *not* immediately update its clock with the offset θ upon receipt of a server response packet. Instead, the client collects several timing samples from each server by completing exchanges at infrequent *polling intervals* (on the order of seconds or minutes). The length of the polling interval is determined by an adaptive randomized *poll process* [27, Sec. 13]. The poll p is a field on the NTP packet, where [27] allows $p \in \{4, 5, .., 17\}$, which corresponds to a polling interval of about 2^p (*i.e.,* 16 seconds to 36 hours).

Once the client has enough timing samples from a server, it computes the jitter ψ. First, it finds the offset value θ^* corresponding to the sample of lowest delay δ^* from the eight most recent samples, and then takes jitter ψ as

$$\psi^2 = \tfrac{1}{k-1}\sum_{i=1}^{k}(\theta_i - \theta^*)^2 \tag{3}$$

Typically, $4 \le k \le 8$. A client considers updating its clock only if it gets a stream of k timing samples with low delay δ and jitter ψ. This is called `TEST11`.[2]

TEST11: Check that the *root distance* Λ does not exceed MAXDIST = 1.5 seconds. Λ is proportional[3] to:

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

The root delay Δ, root dispersion E and precision ρ are from fields in the server's mode 4 response packet (Figure 4). Precision ρ is the quality of the system's local clock; $\rho = 12$ implies $2^{-12} = 244\mu$s precision.

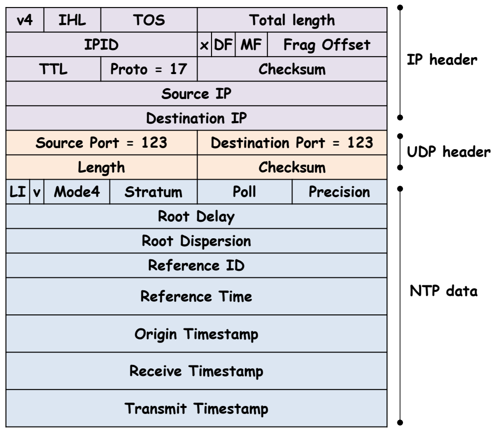

Fig. 4. NTP server response packet (mode 4). (Client queries have the same format, but with mode field set to 3. Symmetric mode uses mode 1 or 2. Broadcast mode uses mode 5).

After each exchange, the client chooses a *single* server to which it synchronizes its local clock. This decision is made adaptively by a set of *selection*, *cluster*,

[2] A single server response packet is sufficient to set time on a SNTP ("simple NTP") client, but a stream of self-consistent packets is required for full NTP.

[3] The exact definition of Λ differs slightly between RFC5905 [27, Appendix A.5.5.2] and the latest version of ntpd.

combine and *clock discipline* algorithms [27, Sec. 10-12]. Importantly, these algorithms can also decide *not* to update the client's clock; in this case, the clock runs without input from NTP.

Implementation vs. Specification. RFC5905 [27] specifies NTP version 4, and its "reference implementation" is ntpd [40]. David Mills, the inventor of NTP, explains [28] the "relationship between the published standard and the reference implementation" as follows: "It is tempting to construct a standard from first principles, submit it for formal verification, then tell somebody to build it. Of the four generations of NTP, it did not work that way. Both the standard and the reference implementation were evolved from an earlier version... Along the way, many minor tweaks were needed in both the specification and implementation..." For this reason, we consider both ntpd and the specification in RFC5905.

3 The Client/Server Protocol in RFC5905

We now argue that the client/server datagram protocol in RFC5905 is underspecified and flawed. RFC5905 mentions the protocol in two places: in its main body (Section 8) and in a pseudo-code listing (Appendix A). Because the two mentions are somewhat contradictory, we begin with an overview of the components of NTP's datagram protocol, and then present its specification in Appendix A of RFC5905, and in the prose of Section 8 of RFC5905.

3.1 Components of NTP's datagram protocol.

NTP uses the *origin timestamp* field of the NTP packet to prevent off- and on-path attacks. (Recall from Figure 2 that an off-path attacker can spoof IP packets but cannot eavesdrop on its target's NTP traffic, while an on-path attacker can eavesdrop, inject, spoof, and replay packets, but cannot drop, delay, or tamper with legitimate traffic.) Whenever a client queries its server, the client records the query's sending time T_1 in a local *state variable* [27] named "`xmt`". The client then sends T_1 in the *transmit timestamp* of its client query (Figure 4). Upon receipt of the query, the server learns T_1 and copies it into the *origin timestamp* field of its server response (Figure 4). When the client receives the server response, it performs `TEST2`:

```
receive()
    if (pkt.T3 == 0 or   # fail test3
         pkt.T3 == org): # fail test1
        return

    synch = True
    if !broadcast:
        if pkt.T1 == 0:  # fail test3
            synch = False
        elif pkt.T1 != xmt:  # fail test2
            synch = False

    org = pkt.T3
    rec = pkt.time_received
    if (synch):
        process(pkt)
```

Fig. 5. Pseudocode for the receive function, RFC5905 Appendix A.5.1.

`TEST2`: The client checks that the origin timestamp T_1 on the server response matches the client's time upon sending the query, as recorded in the client's local state variable `xmt`.

The origin timestamp is therefore a nonce that the client must check (with `TEST2`) before it accepts a response.[4] An off-path attacker cannot see the origin timestamp (because it cannot observe the exchange between client and server), and thus has difficulty spoofing a server response containing a valid origin timestamp. Indeed, the origin timestamp looks somewhat random to the off-path attacker. Specifically, its first 32 bits are seconds, and the last 32 bits are subseconds (or fractional seconds). The first 32 bits appear slightly random because the off-path attacker does not know the exact moment that the client sent its query; indeed, Appendix A of RFC5905 has a comment that says "While not shown here, the reference implementation randomizes the poll interval by a small factor" and the current ntpd implementation randomizes the polling interval by 2^{p-4} seconds when poll $p > 4$. Moreover, the last 32 bits also appear somewhat random because RFC5905 requires a client with a clock of precision ρ randomize the $(32 - \rho)$- lowest-order bits of the origin timestamp.

The origin timestamp thus is analogous to source port randomization in TCP/UDP, sequence number randomization in TCP, *etc.* When NTP packets are cryptographically authenticated with a MAC, this nonce also provides some replay protection: even an on-path attacker cannot replay a packet from an earlier polling interval because its origin timestamp is now stale.

NTP also has mechanisms to prevent replays within the same polling interval. These are needed because an NTP client continuously listens to network traffic, even when it has no outstanding (*i.e.,* unanswered) queries to its servers. Whenever a client receives a server response packet, it records the transmit timestamp field from the packet in its `org` state variable. The client uses the following test to reject duplicate server response packets:

`TEST1`: The client checks that the transmit timestamp field T_3 of the server response is *different* from the value in the client's `org` state variable.
The client deals with the duplicates of the client's query as follows:

Clear `xmt`: If a server response passes `TEST2`, the client sets its local `xmt` state variable to zero.

Suppose the server receives two identical client queries. The server would send responses to both (because NTP servers are stateless [27]). If the client cleared `xmt` upon receipt of the first server response, the second server response packet will be rejected (by `TEST2`) because its origin timestamp is non-zero. At this point, one might worry that an off-path attacker could inject a packet with origin timestamp set to zero. But, `TEST3` should catch this:

`TEST3`: Reject any response packet with origin, receive, or transmit timestamp T_1, T_2, T_3 set to zero.

3.2 Query replay vulnerability in Appendix A of RFC5905.

Pseudocode from Appendix A of RFC5905 (see Figure 5) handles the processing of received packets of *any* mode, including server mode packets (mode 4),

[4] Note that ntpd does not randomize the UDP source port to create an additional nonce; instead, all NTP packets have UDP source port 123.

broadcast mode packets (mode 5), and symmetric mode packets (mode 1 or 2). Importantly, this pseudocode requires a host to always listen to and process incoming packets. This is because some NTP modes (*e.g.,* broadcast) process unsolicited packets, and RFC5905 suggest that all modes use the same codepath. We shall see that this single codepath creates various security problems.

On-path query replay vulnerability. The pseudocode in Figure 5 is vulnerable to replays of the client's query. Suppose a client query is replayed to the server. Then, the server will send two responses, each with a valid origin timestamp field (passing `TEST2`) and each with a different transmit timestamp field (passing `TEST1`). The client will accept both responses. Our experiments show that replays of the client query harm time synchronization; see Appendix C.

3.3 Zero-0rigin timestamp vulnerability in RFC5905 prose.

Meanwhile, we find the following in Section 8 of RFC5905:

> Before the xmt and org state variables are updated, two sanity checks are performed in order to protect against duplicate, bogus, or replayed packets. In the exchange above, a packet is duplicate or replay if the transmit timestamp t3 in the packet matches the org state variable T3. A packet is bogus if the origin timestamp t1 in the packet does not match the xmt state variable T1. In either of these cases, the state variables are updated, then the packet is discarded. To protect against replay of the last transmitted packet, the xmt state variable is set to zero immediately after a successful bogus check.

This text describes `TEST1` and `TEST2`, but what does it mean to update the state variables? Comparing this to the pseudocode in Appendix A of RFC5905 (Figure 5 lines 13-14) suggests that this means updating `org` and `rec` upon receipt of any packet (including a bogus one failing `TEST2`), but not the `xmt` state variable.[5] Next, notice that the quoted text does not mention `TEST3`, which rejects packets with a zero-0rigin timestamp. Thus, we could realize the quoted text as pseudocode by deleting lines 8-9 of Figure 5. Finally, notice that the quote suggests clearing `xmt` if a received packet passes `TEST2`. Thus, we could add the following after line 11 of Figure 5 (with lines 8-9 deleted):

```
else: xmt = 0
```

However, if `xmt` is cleared but `TEST3` is not applied, we have:

Zero-0rigin Timestamp Attack. The zero-0rigin timestamp vulnerability allows an off-path attacker to hijack an unauthenticated client/server association and shift time on the client.

[5] Indeed, suppose we did update the `xmt` variable even after receipt of a bogus packet that fails `TEST2`, with the bogus origin timestamp in the received packet. In this case, we would be vulnerable to a *chosen-origin-timestamp attack*, where an attacker injects a first packet with an origin timestamp of the their choosing. The injected packet fails `TEST2` and is dropped, but its origin timestamp gets written to the target's local `xmt` variable. Then, the attacker injects another packet with this same origin timestamp, which passes `TEST2` and is accepted by the target.

The attacker sends its target client a spoofed server response packet, spoofed with the source IP address of the target's server.[6] The spoofed server response packet has its origin timestamp T_1 set to zero, and its other timestamps T_2, T_3 set to bogus values designed to convince the client to shift its time. The target will accept the spoofed packet as long is it does not have an outstanding query to its server. Why? If a client has already received a valid server response, the valid response would have cleared the client's `xmt` variable to zero. The spoofed zero-0rigin packet is then subjected to `TEST2`, and its origin timestamp (which is set to zero) will be compared to the `xmt` variable (which is also zero). `TEST3` is never applied, and so the spoofed zero-0rigin packet will be accepted.

Suppose that the attacker wants to convince the client to change its clock by x years. How should the attacker set the timestamps on its spoofed packet? The origin timestamp is set to $T_1 = 0$ and the transmit timestamp T_3 is set to the bogus time $\text{now} + x$. The destination timestamp T_4 (not in the packet) is $\text{now} + d$, where d is the latency between the moment when the attacker sent its spoofed packet and the moment the client received it. Now, the attacker needs to choose the receive timestamp T_2 so that the delay δ is small. (Otherwise, the spoofed packet will be rejected because it fails `TEST11` (Section 2).) Per equation (1), if the attacker wants delay $\delta = d$, then T_2 should be:

$$T_2 = \delta + T_3 - (T_4 - T_1) = d + \text{now} + x - (\text{now} + d + 0) = x$$

The offset is therefore $\theta = x - \frac{d}{2}$. If the attacker sends the client a stream of spoofed packets with timestamps set as described above, their jitter ϕ is given by the small variance in d (since x is a constant value). Thus, if the attacker sets root delay $\varDelta$, root dispersion E and precision ρ on its spoofed packets to be tiny values, the packet will pass `TEST11` and be accepted. This vulnerability is actually present in the current version of ntpd. We discuss how we executed it (CVE-2015-8138) against ntpd in Appendix A.1.

4 Leaky Control Queries

Thus far, we have implicitly assumed that the timestamps stored in a target's state variables are difficult for an attacker to obtain from off-path. However, we now show how they can be learned from off-path via NTP control queries. Interestingly, the control queries we use are not mentioned at all in the latest NTP specification in RFC5905 [27]. However, they are specified in detail in

[6] As observed by [21], hosts respond to unauthenticated mode 3 queries from arbitrary IP addresses by default. The mode 4 response (Figure 4) has a *reference ID* field that reveals the IPv4 address of the responding host's time server. Thus, our off-path attacker sends its target a (legitimate) mode 3 query, and receives in response a mode 4 packet, and learns the target's server from its reference ID. Moreover, if the attacker's shenanigans cause the target to synchronize to a different server, the attacker can just learn the IP of the new server by sending the target a new mode 3 query. The attacker can then spoof packets from the new server as well.

Appendix B of the obsolete RFC1305 [25] from 1992, and are also specified in a new IETF Internet draft [26]. They have been part of ntpd since at least 1999.[7] NTP's UDP-based control queries are notorious as a vector for DDoS amplification attacks [11, 20]. These DoS attacks exploit the *length* of the UDP packets sent in response to NTP's mode 7 `monlist` control query, and sometimes also NTP's mode 6 `rv` control query. Here, however, we will exploit their *contents*.

The leaky control queries. We found control queries that reveal the values stored in the `xmt` (which stores T_1 per Figure 3) and `rec` (which stores T_4) state variables. First, launch the `as` control query to learn the association ID that a target uses for its server(s). (Association ID is a randomly assigned number that the client uses internally to identify each server [25].) Then, the query `rv assocID org` reveals the value stored in `xmt` (*i.e.,* expected origin timestamp T_1 for that server). Moreover, `rv assocID rec` reveals the value in `rec` (*i.e.,* the destination timestamp T_4 for the target's last exchange with its server).

Off-path timeshifting via leaky origin timestamp. If an attacker could continuously query its target for its expected origin timestamp (*i.e.,* the `xmt` state variable), then all bets are off. The off-path attacker could spoof bogus packets that pass `TEST2` and shift time on the target. This is CVE-2015-8139.

Off-path timeshifting attack via interleaved pivot. NTP's interleaved mode is designed to provide more accurate time synchronization. Other NTP modes use the 3-bit *mode* field in the NTP packet (Figure 4) to identify themselves (*e.g.,* client queries use mode 3 and server responses use mode 4). The interleaved mode, however, does not. Instead, a host will *automatically* enter interleaved mode if it receives a packet that passes *Interleaved `TEST2`. Interleaved `TEST2`* checks that the packet's *origin timestamp* field T_1 matches `rec` state variable, which stores T_4 from the previous exchange. Importantly, there is no codepath that allows the host to exit interleaved mode. Appendix A.2 shows that this leads to an extremely low-rate DoS attack that works even in the absence of leaky control queries. This is CVE-2016-1548.

Now consider an off-path attacker that uses NTP control queries to continuously query for `rec`. This attacker can shift time on the client by using its knowledge of `rec` to (1) spoof a single packet passing 'interleaved `TEST2`' that pivots the client into interleaved mode, and then (2) spoof a stream of self-consistent packets that pass 'interleaved `TEST2`' and contain bogus timing information. We have confirmed that this attack works on ntpd v4.2.8p6.

Recommendation: Block control queries! By default, ntpd allows the client to answer control queries sent by any IP in the Internet. However, in response to `monlist`-based NTP DDoS amplification attacks, best practices recommend configuring ntpd with the `noquery` parameter [40]. While `noquery` should block all control queries, we suspect that `monlist` packets are filtered by middleboxes, rather than by the `noquery` option, and thus many "patched" systems remain vulnerable to our attacks. Indeed, the openNTPproject's IPv4 scan during the week of July 23, 2016 found 705,183 unique IPs responding to `monlist`. Mean-

[7] `https://github.com/ntpsec/ntpsec/blob/PRE_NT_991015/ntpq/ntpq.c`

while, during the same week we found a staggering 3,964,718 IPs responding to the `as` query.[8] The control queries we exploit likely remain out of firewall blacklists because (1) they are undocumented in RFC5905 and (2) are thus far unexploited. As such, we suggest that either (1) `noquery` be used, or (2) firewalls block *all* mode 6 and mode 7 NTP packets from unwanted IPs.

5 Measuring the Attack Surface

We use network measurements to determine the number of IPs in the wild that are vulnerable to our off-path attacks. We start with zmap [15] to scan the entire IPv4 address space (from July 27 - July 29, 2016) using NTP's `as` control query and obtain responses from 3,964,718 unique IPs. The scan was broken up into 254 shards, each completing in 2-3 minutes and containing 14,575,000 IPs. At the completion of each shard, we run a script that sends each responding IP the sequence of queries shown below.

```
rv ‘associd’
rv ‘associd’ org
rv ‘associd’ rec
rv
mode 3 NTPv4 query
```

These queries check for leaky origin and destination timestamps, per Section 4, and also solicit a regular NTP server response packet (mode 4). Our scan did not modify the internal state of any of the queried systems. We solicit server responses packets using RFC5905-compliant NTP client queries (mode 3), and RFC1305-compliant mode 6 control packets identical to those produced by the standard NTP control query program ntpq. We obtained a response to at least one of the control queries from 3,822,681 (96.4%) of the IPs responding to our `as` scan of IPv4 address space. We obtained server response packets (mode 4) from 3,274,501 (82.6%) of the responding IPs. Once the entire scan completed on July 29, 2016, we identified all the stratum 1 servers (from the `rv` and mode 4 response packets), and send each the NTP control query `peers` using ntpq; we obtained responses from 3,586 (76.6%) IPs out of a total of 4,683 IPs queried. (We do this to check if any stratum 1 servers have symmetric peering associations, since those that do could be vulnerable to our attacks.)

5.1 State of crypto.

The general wisdom suggests that NTP client/server communications are typically not cryptographically authenticated; this follows because (1) NTP uses pre-shared symmetric keys for its MAC, which makes key distribution cumbersome [5], and (2) NTP's Autokey [16] protocol for public-key authentication is widely considered to be broken [34]. We can use our scan to validate the general wisdom, since `as` also reveals a host's ‘authentication status’ with each of its servers or peers. Of 3,964,718 IPs that responded to the `as` command, we find merely 78,828 (2.0%) IPs that have *all* associations authenticated. Meanwhile, 3,870,933 (97.6%) IPs have *all* their associations unauthenticated. We

[8] To avoid being blacklisted, we refrained from sending `monlist` queries.

find 93,785 (2.4%) IPs have at least one association authenticated. For these hosts, off-path attacks are more difficult but not infeasible (especially if *most* of the client's associations are unauthenticated, or if the authenticated associations provide bad time, *etc.*).

5.2 Leaky origin timestamps.

Table 1. Hosts leaking origin timestamp.

Total	unauthenticated	Stratum 2-15	good timekeepers
3,759,832	3,681,790	2,974,574	2,484,775

Of 3,964,718 IPs responding to the `as` query, a staggering 3,759,832 (94.8%) IPs leaked their origin timestamp. (This is a significantly larger number than the 705,183 IPs that responded to a `monlist` scan of the IPv4 space by the openNTPproject during the same week, suggesting that many systems that have been 'patched' against NTP DDoS amplification [11, 20] remain vulnerable to our leaky-origin timestamp attack.)

But how many of these leaky hosts are vulnerable to off-path timeshifting attacks described in Section 4? Our results are summarized in Table 1. First, we find that only 78,042 (2.1%) of the IPs that leak `org` to us have authenticated *all* associations with their servers, leaving them out of the attackable pool. Next, we note that stratum 1 hosts are not usually vulnerable to this attack, since they sit a the root of the NTP hierarchy (see Section 2) and thus don't take time from any server. The only exception to this is the stratum 1 servers that have symmetric peering associations. Combining data from `rv` and mode 3 responses, we find the stratum of the remaining 3,681,790 (97.9%) leaky IPs. We combine this information with the output of the `peers` command, which reveals the 'type of association' each host uses with its servers and peers. Of the 4,608 (0.1%) stratum 1 servers, *none* have symmetric peering associations. Thus, we do not find *any* vulnerable stratum 1 servers.

On the other hand, there are 2,974,574 (80.8%) stratum 2-15 IPs that leak their origin timestamp and synchronize to at least one unauthenticated server. These are all vulnerable to our attack. We do not count 601,043 (16.3%) IPs that have either (1) stratum 0 or 16 (unsynchronized), OR (2) conflicting stratums in `rv` and server responses (mode 4). Finally, we check if these 3M vulnerable IPs are 'functional' or are just misconfigured or broken systems by using data from our mode 3 query scan to determine the quality of their timekeeping. We found that 2,484,775 (83.5%) of these leaky IPs are good timekeepers—their absolute offset values were less than 0.1 sec.[9] Of these, we find 490,032 (19.7%) IPs with stratum 2. These are good targets for attack, so that the impact of the attack trickles down the NTP stratum hierarchy.

5.3 Zero-0rigin timestamp vulnerability.

[9] We compute the offset θ using equation (2), with T_1, T_2, T_3 from the packet timestamps and T_4 from the frame arrival time of the mode 4 response packet .

Table 2. Hosts leaking zero-0rigin timestamp.

Total	unauthenticated	Stratum 2-15	good timekeepers
1,269,265	1,249,212	892,672	691,902

The zero-0rigin timestamp vulnerability was introduced seven years ago in ntpd v4.2.6 (Dec 2009), when a line was added to clear `xmt` after a packet passes `TEST2`.[10] (This is Line 18 in Figure 11.)Thus, one way to bound the attack surface for the zero-0rigin timestamp vulnerability is to use control queries as measurement side-channel. We consider all our origin-timestamp leaking hosts, and find the ones that leak a timestamp of zero. Of 3,759,832 (94.8%) origin-leaking IPs, we find 1,269,265 (33.8%) IPs that leaked a zero-0rigin timestamp. We scrutinize these hosts in Table 2 and find $\approx 700K$ interesting targets. Importantly, however, that this is likely an underestimate of the attack surface, since the zero-0rigin vulnerability does *not* require the exploitation of leaky control queries.

5.4 Interleaved pivot vulnerability.

Table 3. Hosts leaking `rec` and zero-0rigin timestamps. (Underestimates hosts vulnerable to the interleaved pivot timeshifting attack.)

Total	unauthenticated	Stratum 2-15	good timekeepers
1,267,628	1,247,656	893,979	691,393

The interleaved pivot DoS vulnerability (Appendix A.2)was introduced in the same version as the zero-0rigin timestamp vulnerability. Thus, the IPs described in Section 5.3 are also vulnerable to this attack.

Next, we check which IPs are vulnerable to the interleaved pivot *timeshifting attacks* (Section 4). These hosts must (1) leak the `rec` state variable and (2) use a version of ntpd later than 4.2.6. Leaks of `rec` are also surprisingly prevalent: 3,724,465 IPs leaked `rec` (93.9% of the 4M that responded to `as`). These could be vulnerable if they are using ntpd versions post v4.2.6. We cannot identify the versions of all of these hosts, but we do know that hosts that also leak zero as their expected origin timestamp are using versions post v4.2.6. We find 1,267,265 (34%) such IPs and scrutinize them in Table 3.

6 Securing the Client/Server Protocol.

We now move beyond identifying attacks and *prove security* for modified client/server datagram protocols for NTP.

6.1 Protocol descriptions.

Our protocol. Figure 6,7 present our new client/server protocol that provides 32-bits of randomization for the origin timestamp used in `TEST2`.

Clients use the algorithm in Figure 6 to process received packets. While the client continues to listen to server response packets (mode 4) even when it does not have an outstanding query, this receive algorithm has several features that

[10] See Line 1094 in `ntp_proto.c` in `https://github.com/ntp-project/ntp/commit/fb8fa5f6330a7583ec74fba2dfb7b6bf62bdd246`.

```
def client_receive_mode4( pkt ):

    server = find_server(pkt.srcIP)

    if (server.auth == True and
       pkt.MAC is invalid):
        return           # bad MAC

    if pkt.T1 != server.xmt:
        return           # fail test2

    server.xmt = randbits(64)  # clear xmt
    server.org = pkt.T3        # update state variables
    server.rec = pkt.receive_time()
    process(pkt)
return
```

Fig. 6. Pseudocode for processing a response. We also require that the `xmt` variable be initialized as a randomly-chosen 64-bit value, *i.e.,* `server.xmt = randbits(64)`, when ntpd first boots.

```
def client_transmit_mode3_e32( precision ):

    r = randbits(precision)
    sleep for r*(2**(- precision)) seconds

    # fuzz LSB of xmt
    fuzz = randbits(32 - precision)
    server.xmt = now ^ fuzz

    # form the packet
    pkt.T1 = server.org
    pkt.T2 = server.rec
    pkt.T3 = server.xmt
    ... # fill in other fields

    if server.auth == True:
        MAC(pkt)    #append MAC

    send(pkt)
return
```

Fig. 7. This function is run when the polling algorithm signals that it is time to query `server`. If `server.auth` is set, then `pkt` is authenticated with a MAC.

differ from RFC5905 (Figure 5).First, when a packet passes `TEST2`, we clear `xmt` by setting it to a random 64-bit value, rather than to zero. We also require that, upon reboot, the client initializes its `xmt` values for each server to a random 64-bit value. Second, `TEST2` alone provides replay protection and we eliminate `TEST1` and `TEST3`. (`TEST3` is not needed because of how `xmt` is cleared. Eliminating `TEST3` is also consistent with the implementation in ntpd versions after v4.2.6.)

Clients use the algorithm in Figure 7 to send packets. Recall that the first 32 bits of the origin timestamp are seconds, and the last 32 bits are subseconds. First, a client with a clock of precision ρ put a $(32-\rho)$-bit random value in the $(32-\rho)$ lowest order bits. Next, the client obtains the remaining ρ bits of entropy by randomizing the packet's sending time. When the polling algorithm indicates that a query should be sent, the client sleeps for a random subsecond period in $[0, 2^{-\rho}]$ seconds, and then constructs the mode 3 query packet. We therefore obtain 32 bits of entropy in the expected origin timestamp, while still preserving the semantics of NTP packets—the mode 4 packet's origin timestamp field (Figure 4) still contains T_1 (where T_1 is as defined in Figure 3).

Notice that this protocol only modifies the client, and is fully backwards-compatible with today's stateless NTP servers:

Stateless server algorithm. Today's NTP servers are stateless, and so do not keep `org` or `xmt` state variables for their clients. Instead, upon receipt of client's mode 3 query, a server immediately sends a mode 4 response packet with (1) origin timestamp field equal to the transmit timestamp field on the query, (2) receive timestamp field set to the time that the server received the query, and (3) transmit timestamp field to the time the server sent its response.

Chronyd/openNTPd protocol. The chronyd and openNTPd implementations also use a client/server protocol that differs from the one in RFC5905. This protocol just sets the expected origin timestamp to be a random 64-bit nonce (see Figure 8). While this provides 64-bits of randomness in the origin timestamp, it breaks the semantics of the NTP packet timestamps, because the server response packet no longer contains T_1 as defined in Figure 3. (Instead, the client must additionally retain T_1 in local state variable `server.localxmt`.) This means that the chrony/openNTPd protocol *cannot* be used for NTP's symmetric mode (mode 1/2), but our protocol (which preserves timestamp semantics) can be used for symmetric mode. (See footnote 18.)

```
def client_transmit_mode3_e64( precision ):

    # store the origin timestamp locally
    server.localxmt = now

    # form the packet
    server.xmt = randbits(64) #64-bit nonce
    pkt.T1 = server.org
    pkt.T2 = server.rec
    pkt.T3 = server.xmt
    ... # fill in other fields

    if server.auth == True:
        MAC(pkt)    #append MAC

    send(pkt)
return
```

Fig. 8. Alternate client/server protocol used by chronyd/openNTPd, that randomizes all 64-bits of the origin timestamp. This function is run when the polling algorithm signals that it is time to query `server`.

Security. Both our protocol (Figures 6,7) and the chronyd/openNTPd protocol (Figures 6,8) can be used to protect client/server mode from off-path attacks (when NTP packets are unauthenticated) and on-path attacks (when NTP packets are authenticated with a secure message authentication code (MAC)[11].) Security holds as long as (1) all randomization is done with a cryptographic pseudorandom number generator (RNG), rather than the weak `ntp_random()` function currently used by ntpd [3], (2) the expected origin timestamp is not leaked via control queries, and (3) NTP strictly imposes $k = 4$ or $k = 8$ as the minimum number of consistent timing samples required before the client considers updating its clock. The last requirement is needed because 32-bits of randomness, alone, is not sufficient to thwart a determined attacker. However, by requiring k consistent timing samples in a row, the attacker has to correctly guess about $32k$ random bits (rather than just 32 random bits). Fortunately, because of TEST11 (see Section 2), ntpd already requires $k \geq 4$ *most* of the time.

To obtain these results, we first develop a cryptographic model for security against off- and on-path NTP attacks (Section 6.2). We then use this model prove security for off-path attacks (Section 6.3) and on-path attacks (Section 6.4), both for our protocol, and for the chronyd/openNTPd protocol.

[11] RFC5905 specifies MD5(key||message) for authenticating NTP packets, but this is not a secure MAC [7]. We are currently in the processes of standardizing a new secure MAC for NTP [23].

6.2 Security Model.

Our model, which is detailed in Appendix F, is inspired by prior cryptographic work that designs synchronous protocols with guaranteed packet delivery [6, 18]. However, unlike these earlier models, we consciously omit modeling the more powerful MiTM who can drop, modify, or delay packets (see Section 1 and Figure 2). We assume instead that the network delivers all packets sent between the ℓ honest parties $\mathcal{P}_1$,...,$\mathcal{P}_\ell$. We also assume that the network does not validate the source IP in the packets it transits, so that the attacker can *spoof* packets. Honest parties experience a delay ϱ before their packets are delivered, but the attacker can win every race condition.

The network orchestrates execution of several NTP exchanges (akin to the 'environment' in the universal composability framework [8]) through the use of a *transcript* that stipulates (1) which parties engage in two-message client/server exchanges with each other, (2) when they engage in each exchange, and (3) the times t_c and t_s on the local clocks of the client and server respectively during each exchange. We require security over *all* possible transcripts. This means, as a corollary, that the attacker can choose the optimal transcript for her to attack, including having control over the local clocks of *all* honest parties. An on-path attacker can see every packet sent between honest parties, while an off-path attacker can only see the packet's IP header. (See Figure 2.) Clients update their local state which includes (1) the set of servers they are willing to query, (2) the state variables (*e.g.,* $\texttt{xmt}_j$, $\texttt{org}_j$, $\texttt{rec}_j$) for each server $\mathcal{P}_j$, and (3) timing samples from their k most recent exchanges with each server. Then:

Definition 61 (Soundness (Informal))**.** NTP is (k, ϵ)-*sound* on transcript $\texttt{ts}$ if for all *resource-bounded* attackers $\mathcal{A}$ and all parties $\mathcal{P}_i$ who do not query $\mathcal{A}$ as an NTP server, $\mathcal{P}_i$ has k consecutive timing samples from one of its trusted servers that have been modified by $\mathcal{A}$ with probability ϵ. The probability is over the randomness of all parties.

We parameterize by k because NTP has mechanisms that prevent synchronization until the host has a stream of consistent timing samples from a server or peer most likely to represent accurate time. TEST11 enforces this, for example, by requiring jitter $\psi < 1.5$ seconds. (See Section 2).

But how should we parameterize k? One idea is $k = 8$, because TEST11 depends on the jitter ψ which is computed over at most eight consecutive timing samples (equation 3). $k = 8$ is also consistent with pseudocode in Appendix A.5.2 of RFC 5905; this pseudocode describes the algorithm used for clock updates and includes the comment "select the best from the latest eight delay/offset samples". This may be too optimistic though, because we have observed that ntpd v4.2.6 requires only $k = 4$ before it updates its clock upon reboot. ntpd v4.2.8p6 requires only one sample upon reboot but this is a bug (CVE-2016-7433); see Appendix D. Thus, we consider $k \in \{1, 4, 8\}$.

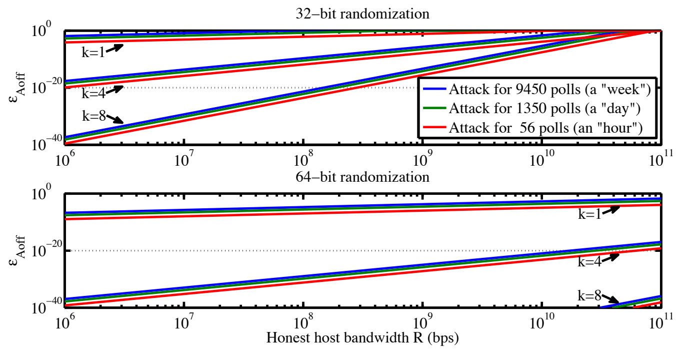

Fig. 9. Success probability of off-path attacker per Theorem 61: (Top) for Figures 6,7 and (Bottom) for Figures 6,8. $\tau \in \{56, 1350, 9540\}$ is the number of polling intervals attacked. We assume one server ($s = 1$) and latencies of at most $\varrho = 1$ second.

6.3 Security analysis: Unauthenticated NTP & off-path attacks.

We now discuss the security guarantees for the protocols described in Appendix 6. We start by considering off-path attacks. At a high level, our protocol and the chronyd/openNTPd protocols thwart off-path attackers due to the unpredictability of the origin timestamp. Preventing off-path attacks is the best we can hope for when NTP is unauthenticated, since on-path attackers (that can observe the expected origin timestamp per Figure 2) can trivially spoof unauthenticated server responses.

We assume that honest parties can send and receive packets at rate at most R bits per second (bps). The network imposes latencies of $\leq \varrho$ for packets sent by any honest party. The polling interval is 2^p, where RFC5905 constrains $p \leq 17$. Let $\mathit{off}\mathcal{A}$ denote the off-path attackers and let ts be any transcript that involves ℓ honest parties, τ the maximum number of exchanges involving any single client-server pair, k is the minimum number of consistent timing samples needed for a clock update and s the maximum number of trusted servers per client. Also let Adv(RNG) denote the maximum advantage that any attacker with $\mathit{off}\mathcal{A}$'s resource constraints has of distinguishing a pseudorandom number generator from a random oracle. Then both protocols satisfy the following:

Theorem 61. *Suppose NTP is unauthenticated. Let* $\mathit{off}\mathcal{A}$*,* k*,* ϱ*,* p*,* R*,* ts*,* ℓ*,* s*,* τ *and* $Adv(RNG)$ *be as described above. Then the protocol Figures 6 7 is* $(k, \epsilon_{\mathit{off}\mathcal{A}})$*-sound on transcript* ts *with*

$$\epsilon_{\mathit{off}\mathcal{A}} = Adv(RNG) + (k+1)s\tau \cdot \left[\frac{2^{-32}kR\varrho}{360}\right]^k \quad (5)$$

[12]*And the protocol in Figure 6 8 is* $(k, \epsilon_{\mathit{off}\mathcal{A}})$*-sound on transcript* ts *with*

$$\epsilon_{\mathit{off}\mathcal{A}} = Adv(RNG) + (k+1)s\tau \cdot \left[\frac{2^{p-64}kR}{720}\right]^k \quad (6)$$

Figure 9 plots $\epsilon_{\mathit{off}\mathcal{A}}$ versus the bandwidth R at honest parties for $k \in \{1, 4, 8\}$ and different values of τ, where τ is the number of polling intervals for which the attacker launches his attack. We do this both for our protocol in Figure 7 (Figure 9(top)) and the chronyd/openNTPd protocol in Figure 8 (Figure 9(bottom)).Since hosts typically use a minimum poll value $p_{\min} = 6$, the values $\tau = (9450, 1350, 56)$ in Figure 10 correspond to attacking one (week, day, hour) of $2^{p_{\min}} = 64$ second polling intervals. We also assume one server $s = 1$, overestimate network latencies as $\varrho = 1$ second, overestimate poll p in equation (6) as $p = 17$. We assume a good RNG so Adv(RNG) is negligible.

$k = 4$ *is sufficient with 32-bits of randomness.* Recall that k is the minimum number of consistent timing samples needed for a clock update. Figure 9(top) indicates that $k = 4$ suffices for our protocol (that randomizes the 32-bit subsecond granularity of the expected origin timestamp). Even if an off-path attacker attacks for a week, his success probability remains less than 0.1% as long as $k = 4$ and the target accepts packets at bandwidth $R = 5$ Gbps or less. When the attacker attacks for an hour, the target's bandwidth must be $R \approx 19$ Gbps for a 0.1% success probability. To put this in context, endhosts typically send < 10 NTP packets per *minute*, and even the large stratum 1 timeservers operated by NIST process queries at an average rate of 21 Gbps [37]. Therefore, it seems unlikely that an attacker could attack for hours or days without being detected. If more security is needed, we could take $k = 8$, which requires a bandwidth of $R \approx 40$ Gbps for a one-hour attack with success probability of 0.1%. Meanwhile, Figure 9(top) suggests that 32-bits of randomness do not suffice to limit off-path attacks when $k = 1$. This should provide further motivation for fixing the ntpd v4.2.8p6 bug that allows $k = 1$ upon reboot (see Appendix D).

$k = 1$ *is sufficient with 64-bits of randomness.* Meanwhile, Figure 10 (bottom) indicates that $k = 1$ suffices for the chronyd/openNTPd protocol that randomizes all 64 bits of the expected origin timestamp. Even if an off-path attacker attacks for a week, his success rate remains less than 0.1% as long as the target's bandwidth is limited to $R = 5$ Gbps. Moreover, when $k = 4$, attacking for a week at 100 Gbps only yields a success probability of 10^{-17}.

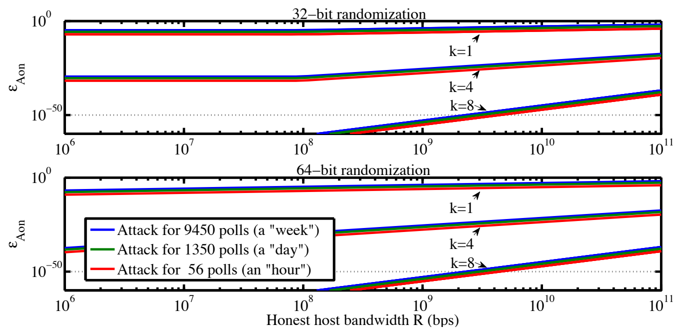

Fig. 10. Success probability of on-path attacker per Theorem 62: (Top) for Figures 6,7; in this case we overestimate the number of legitimate client queries that have identical 32 high-order bits of origin timestamp as $\gamma = 100$. (Bottom) for Figures 6,8. $\tau \in \{56, 1350, 9540\}$ is the number of polling intervals attacked. We assume one server ($s = 1$), latencies of at most $\varrho = 1$ second, MAC of length $2n = 128$ bits and maximum poll value $p = 17$.

6.4 Security analysis: Authenticated NTP & on-path attacks.

Both our protocol (Figure 6,7) and the chronyd/openNTPd protocol (Figure 6,8) thwart on-path attackers when NTP packets are authenticated with a MAC.

We let sending rate R, network latency ϱ, poll p and Adv(RNG) be as before. Let $on\mathcal{A}$ be an on-path attacker, and let ts be any transcript that involves ℓ honest parties, a maximum of s trusted servers per client and a maximum of τ exchanges involving any single client-server pair that replicate any t_c value (up to the second) at most γ times. Let Adv(EU-CMA) be the maximum probability that an attacker with $on\mathcal{A}$'s resource constraints can forge a MAC of length $2n$ under a chosen-message attack. Then both protocols satisfy the following:

Theorem 62. *Suppose NTP is authenticated with a MAC of length $2n$. Let $on\mathcal{A}$, k, ϱ, p, R, ts, ℓ, s, τ, γ, Adv(EU-CMA) and Adv(RNG) be as described above. Then, both protocols are $(k, \epsilon_{on\mathcal{A}})$-sound on transcript ts with*

$$\epsilon_{on\mathcal{A}} \leq Adv(RNG) + (k+1)s\tau(kQ)^k, \qquad (7)$$

[12] Our soundness definition (Appendix 6.2) both allows the off-path attacker to choose the transcript (and thus also the clients local time t_c when it sends the packet) and to see the IP header (only) of the sent packet. Thus, for our protocol (that provides 32-bits of randomness), the off-path attacker essentially knows T_1 up to the second (but not sub-second) granularity. This allows us to claim security even against an off-path attacker that predicts the behavior of a target's polling algorithm (but not her cryptographic random number generators (RNGs)). Some off-path attackers may realistically be able to do this. Consider an off-path attacker that sends a target a 'packet-of-death' that triggers a reboot of ntpd (*e.g.,* CVE-2016-9311 or CVE-2016-7434). Because the attacker knows when ntpd rebooted, it may be able to predict the behavior of its polling algorithm.

where

$$Q = \max\left\{ q_E + \frac{R\varrho \cdot Adv(EU\text{-}CMA)}{360+n}, \frac{2^{p-64}R\varrho}{720+2n} \right\}. \tag{8}$$

where $q_E = 2^{-32}\gamma$ for the protocol in Figures 6 7 and $q_E = 2^{-64}\tau$ for the protocol in Figures 6 8.

To argue about security, we assume a good MAC (like CMAC [23]) so that Adv(EU-CMA) $\approx 2^{-128}$. We overestimate $p = 17$ in equation (8) and $\varrho = 1$ second and plot $\epsilon_{on\mathcal{A}}$ versus R for one server ($s = 1$) and different choices of τ in Figure 10.

With 32-bits of randomness, $k = 4$ is sufficient. Suppose the 32-bits sub-second granularity of the expected origin timestamp is randomized. When R is small, Figure 10(top) indicates that the on-path attacker's success rate is dominated by the first term inside the maximum in equation (8). This corresponds to a successful replay attack, because the client has sent multiple queries with the same expected origin timestamp. Meanwhile, when R is large, second term in the maximum in equation (8) dominates. This corresponds to a successful replay attack, because the client 'cleared' `xmt` to a random 64-bit value that matches an origin timestamp in an earlier query. Again, the attacker's success probability is disconcertingly high when $k = 1$. [13] On the other hand, excellent security guarantees are obtained for $k = 4$, so it is safer to have $k \geq 4$.

With 64-bits of randomness, $k = 4$ is sufficient. Suppose now that the entire 64-bits of the expected origin timestamp is randomized. Now the second term in the maximum in equation (8) always dominates. This again corresponds to a successful replay attack, because the client 'cleared' `xmt` to a random 64-bit value that matches an origin timestamp in an earlier query.

7 Summary and Recommendations

We have identified several vulnerabilities in the NTP specifications both in RFC5905 [27] and in its control query specification in (obsoleted) RFC1305 [25], leading to several working off-path attacks on NTP's most widely used client/server mode (Section 3-4). Millions of IPs are vulnerable our these attacks (Section 5). We present denial-of-service attacks on symmetric mode in Appendix B.

Many of our attacks are possible because RFC5905 recommends that same codepath is used to handle packets from all of NTP's different modes. Our strongest attack, the zero-0rigin timestamp attack (CVE-2015-8139), follows because NTP's client/server mode shares the same codepath as symmetric mode. (In Section B.3, we explain why the initialization of symmetric mode requires

[13] The poor results for $k = 1$ and 32-bits of randomization follow because our model allows the 32 high-order bits of the expected origin timestamp to repeat in at most γ different queries. It might be tempting to dismiss this by assuming $\gamma = 0$, but basing security on this is not a good idea. For example, a system might always boot up thinking that it is January 1, 1970.

that hosts accept NTP packets with origin timestamp set to zero; this leads to the zero-0rigin timestamp attack on client/server mode, where the attacker convinces a target client to accept a bogus packet because its origin timestamp is set to zero.) Similarly, the fact that interleave mode and client/server mode shares the same codepath gives rise to the interleave pivot attack (CVE-2016-1548). Thus, we recommend that different codepaths be used for different modes. This is feasible, since a packet's mode is trivially determined by its *mode* field (Figure 4). The one exception is interleaved mode, so we suggest that interleaved mode be assigned a distinguishing value in the NTP packet.

Our attacks also follow because the NTP specification does not properly respect `TEST2`. We therefore propose a new backwards-compatible client/server protocol that gives `TEST2` the respect it deserves (Section 6.1). We developed a framework for evaluating the security of NTP's client/server protocol and used it to prove that our protocol prevents (1) off-path spoofing attacks on unauthenticated NTP and (2) on-path replay attacks when NTP is cryptographically authenticated with a MAC. We have proved the similar results for a different client/server protocol used by chronyd and openNTPD. (See Section 6.3,6.4.) We recommend that implementations adopt either protocol.

Our final recommendation is aimed at systems administrators. We suggest that firewalls and ntpd clients block *all* incoming NTP control (mode 6,7) and timing queries (mode 1,2 or 3) from unwanted IPs (Section 4), rather than just the notorious `monlist` control query exploited in DDoS amplification attacks.

Acknowledgements.

We are grateful to Jared Mauch for access to the openNTPproject data. We thank the Network Time Foundation and the maintainers of chrony and NTPsec for patching vulnerabilities described here. We also thank Majdi Abbas, Stephen Gray, Ran Canetti, Ethan Heilman, Yossi Gilad, Leonid Reyzin, and Matt Street for useful discussions. This work was supported by the MACS project under NSF Frontier grant CNS-1414119, by NSF grant 1350733, by a Sloan Research Fellowship, and by gifts from Cisco.

References

1. `https://github.com/dfoxfranke/nts`.
2. `https://github.com/mlichvar/chrony/blob/master/ntp_core.c#L908`.
3. `https://github.com/ntp-project/ntp/blob/1a399a03e674da08cfce2cdb847bfb65d65df237/libntp/ntp_random.c`.
4. `https://github.com/philpennock/openntpd/blob/master/client.c#L174`.
5. The NIST authenticated ntp service. `http://www.nist.gov/pml/div688/grp40/auth-ntp.cfm` (Accessed: July 2015), 2010.
6. D. Achenbach, J. Müller-Quade, and J. Rill. Synchronous universally composable computer networks. In *Cryptography and Information Security in the Balkans, (BalkanCryptSec'15)*, pages 95–111, 2015.

7. M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology, CRYPTO 1996*, pages 1–15. Springer, 1996.
8. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE, 2001.
9. R. Clayton, S. J. Murdoch, and R. N. Watson. Ignoring the great firewall of china. In *Privacy Enhancing Technologies*, pages 20–35. Springer, 2006.
10. corbixgwelt. Timejacking & bitcoin: The global time agreement puzzle (culubas blog), 2011. `http://culubas.blogspot.com/2011/05/timejacking-bitcoin_802.html` (Accessed Aug 2015).
11. 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.
12. B. Dowling, D. Stebila, and G. Zaverucha. Authenticated network time synchronization. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 823–840, Austin, TX, Aug. 2016. USENIX Association.
13. H. Duan, N. Weaver, Z. Zhao, M. Hu, J. Liang, J. Jiang, K. Li, and V. Paxson. Hold-on: Protecting against on-path dns poisoning. In *Proc. Workshop on Securing and Trusting Internet Names, SATIN*, 2012.
14. R. Durairajan, S. K. Mani, J. Sommers, and P. Barford. Time's forgotten: Using ntp to understand internet latency. *HotNets'15*, November 2015.
15. 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.
16. 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`.
17. E. Itkin and A. Wool. A security analysis and revised security extension for the precision time protocol. *CoRR*, abs/1603.00707, 2016.
18. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *TCC*, pages 477–498, 2013.
19. J. Klein. Becoming a time lord - implications of attacking time sources. Shmoocon Firetalks 2013: `https://youtu.be/XogpQ-iA6Lw`, 2013.
20. L. Krämer, J. Krupp, D. Makita, T. Nishizoe, T. Koide, K. Yoshioka, and C. Rossow. Amppot: Monitoring and defending against amplification ddos attacks. In *International Workshop on Recent Advances in Intrusion Detection*, pages 615–636. Springer, 2015.
21. A. Malhotra, I. E. Cohen, E. Brakke, and S. Goldberg. Attacking the network time protocol. *NDSS'16*, February 2016.
22. A. Malhotra and S. Goldberg. Attacking NTP's Authenticated Broadcast Mode. *SIGCOMM Computer Communication Review*, April 2016.
23. A. Malhotra and S. Goldberg. *Message Authentication Codes for the Network Time Protocol.* Internet Engineering Task Force (IETF), November 2016. `https://datatracker.ietf.org/doc/draft-ietf-ntp-mac/`.
24. J. Mauch. openntpproject: NTP Scanning Project. `http://openntpproject.org/`.
25. D. Mills. *RFC 1305: Network Time Protocol (Version 3) Specification, Implementation and Analysis.* Internet Engineering Task Force (IETF), 1992. `http://tools.ietf.org/html/rfc1305`.

26. D. Mills and B. Haberman. *draft-haberman-ntpwg-mode-6-cmds-00: Control Messages Protocol for Use with Network Time Protocol Version 4*. Internet Engineering Task Force (IETF), May 2016. `https://datatracker.ietf.org/doc/draft-haberman-ntpwg-mode-6-cmds/`.
27. 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`.
28. D. L. Mills. *Computer Network Time Synchronization*. CRC Press, 2nd edition, 2011.
29. N. Minar. A survey of the NTP network, 1999.
30. T. Mizrahi. A game theoretic analysis of delay attacks against time synchronization protocols. In *Precision Clock Synchronization for Measurement Control and Communication (ISPCS)*, pages 1–6. IEEE, 2012.
31. 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`.
32. N. Moreira, J. Lazaro, J. Jimenez, M. Idirin, and A. Astarloa. Security mechanisms to protect ieee 1588 synchronization: State of the art and trends. In *Precision Clock Synchronization for Measurement, Control, and Communication (ISPCS), 2015 IEEE International Symposium on*, pages 115–120. IEEE, 2015.
33. C. D. Murta, P. R. Torres Jr, and P. Mohapatra. Characterizing quality of time and topology in a time synchronization network. In *GLOBECOM*, 2006.
34. S. Röttger. Analysis of the ntp autokey procedures. Master's thesis, Technische Universitt Braunschweig, 2012.
35. J. Selvi. Bypassing HTTP strict transport security. *Black Hat Europe*, 2014.
36. J. Selvi. Breaking SSL using time synchronisation attacks. *DEFCON'23*, 2015.
37. J. A. Sherman and J. Levine. Usage analysis of the NIST internet time service. *Journal of Research of the National Institute of Standards and Technology*, 121:33, 2016.
38. D. Sibold and S. Roettger. *draft-ietf-ntp-network-time-security: Network Time Security*. Internet Engineering Task Force (IETF), 2015. `http://tools.ietf.org/html/draft-ietf-ntp-network-time-security-08`.
39. 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`.
40. H. Stenn. Securing the network time protocol. *Communications of the ACM: ACM Queue*, 13(1), 2015.
41. H. Stenn. Security notice. `http://support.ntp.org/bin/view/Main/SecurityNotice`, April 27 2016.
42. M. Tatarinov. `http://support.ntp.org/bin/view/Main/NtpBug2952`.
43. N. Weaver, R. Sommer, and V. Paxson. Detecting forged TCP reset packets. In *NDSS*, 2009.

A The Client/Server Protocol in ntpd

We present several ntpd vulnerabilities that stem from ambiguities in RFC5905. Figure 11 is a (simplified) description[14] of the code used for the datagram protocol in ntpd v4.2.8p6 (the most recent release as of mid-April 2016).

Our attacks assume that client/server connections are unauthenticated, which is the default in ntpd and is the most common configuration in the wild (Section 5.1). Appendix A.1 presents our *Zero-0rigin Timestamp vulnerability* (CVE-2015-8138) that allows an off-path attacker to completely hijack an unauthenticated association between a client and its server, shifting time on the client. Appendix A.2 presents our *Interleaved Pivot vulnerability* (CVE-2016-1548), an extremely low-rate off-path denial-of-service attack. Importantly, both vulnerabilities affect ntpd clients operating in default mode, are performed from off-path, and require no special assumptions about the client's configuration. Both have been present in ntpd for seven years, since the first release of ntpd v4.2.6 in December 2009. In Section 4 we combine the interleaved pivot vulnerability with information-leaking NTP control queries and obtain a new off-path timeshifting attack.

```
1  def receive( pkt ):
2      if pkt.T3 == 0:
3          flash |= test3  # fail test3
4      elif pkt.T3 == org
5          flash |= test1  # fail test1
6          return
7      elif broadcast == True:
8          ;           # skip further tests
9      elif interleave == False:
10         if pkt.T1 == 0:
11             xmt = 0
12         elif (xmt == 0 or pkt.T1 != xmt):
13             flash |= test2  # fail test2
14             if (rec !=0 and pkt.T1 == rec):
15                 interleave = True
16             return
17         else:
18             xmt = 0   # pass test2, clear xmt
19     elif (pkt.T1 == 0 or pkt.T2 == 0):
20         flash |= test3     # fail test3
21     elif (rec != 0 and rec != pkt.T1):
22         flash |= test2
23         return  # fail interleave test2
24
25     if interleave == False:
26         rec = pk.receive_time()
27     org = pkt.T3
28
29     if flash == True:
30         return
31     else
32         process( pkt )
33 return
```

Fig. 11. Simplified implementation of the datagram protocol from ntpd v4.2.8p6. The packet will not be processed if the `flash` variable is set. `interleave` variable is set when the host is in interleaved mode. Line 16 was introduced at ntpd v4.2.8p4 and 10-11 at ntpd v4.2.8p5.

A.1 Zero-0rigin timestamp vulnerability.

The zero-0rigin timestamp vulnerability allows an off-path attacker to completely hijack an unauthenticated client/server association and shift the client's time.

[14] Note that the actual ntpd code swaps the names of the `xmt` and `org` state variables; we have chosen the description that is consistent with the RFCs.

Injecting 0rigin packets from off-path. In this attack, the attacker sends a spoofed mode 4 response packet to the target client. The spoofed packet has its origin timestamp T_1 set to zero, and its other timestamps T_2, T_3 set to bogus values designed to convince the target to shift its time, and its source IP set to that of the target's server. (The off-path attacker learns the server's IP via the reference ID, per footnote 6.) Now, consider how the target processes the received spoofed zero-0rigin timestamp packet:

(1) For ntpd v4.2.8p5 or v4.2.8p6, a spoofed zero-0rigin packet will always be accepted, because it passes through lines 10-11, which skip `TEST2` altogether. While the addition of lines 10-11 may seem strange, we suspect that they were added to handle the initialization of NTP's symmetric mode, which shares the same code path as client/server mode. Further discussion is in Appendix B.3.

(2) For ntpd v4.2.6 to v4.2.8p4, lines 10-11 of Figure 11 were absent. Thus, the target will accept the spoofed packet when it does not have an outstanding query to the server. Why? When the target does not have an outstanding query to its server, its `xmt` variable is cleared to zero. Thus, when the spoofed zero-0rigin packet is subjected to `TEST2` (line 12), its origin timestamp (which is zero) will be compared to the `xmt` variable (which is also zero) and be accepted. The vulnerability arises because Figure 11 fails to apply `TEST3`, which rejects packets with zero origin timestamp.[15]

Thus, an off-path attacker can send the target a quick burst of self-consistent zero-0rigin packets with a bogus time, and cause the target to shift its time. The spoofed zero-0rigin packets are always accepted in case (1), and usually accepted in case (2) because the target is unlikely to have outstanding query to its server. (In case (2), this follows because the server is queried so infrequently—NTP's polling intervals are at least 16 seconds long, but are often up to 15 minutes long.) After accepting the burst of zero-0rigin packets, the target immediately shifts its time; in fact, the target shifts its time more quickly than it would under normal conditions, when legitimate responses arrive from the server at the (very slow) NTP polling rate (Section 2).

Experiment. On April 29, 2016, we performed a zero-0rigin timestamp attack on an ntpd v4.2.8p6 client. The target client uses the `-g` option on an operating system that restarts ntpd when it quits.[16] The target is configured to take time

[15] `TEST3` is applied in the fifth clause in Figure 11, but a client will not enter this clause unless it is in interleaved mode.

[16] NTP's has a *panic threshold* that is 1000 seconds (16 mins). If the client gets a time shift that exceeds the panic threshold, the client quits. Thus, at first glance, it seems that that the worst an attacker can do is alter the client's clock by 16 minutes. However, as noted in [21], this panic behavior can be exploited. ntpd has a `-g` option that allows a client to ignore the *panic threshold* when it reboot; `-g` is the default ntpd configuration on many OSes including CoreOS Alpha (1032.1.0), Debian 8.2.0, Arch Linux 2016.05.01, etc. Moreover, many operating systems uses process supervisors (*e.g.,* systemd), which can be configured to automatically restart any daemons that quit. (This behavior is the default in CoreOS and Arch Linux. It is likely to become the default behavior in other systems as they migrate legacy init scripts to systemd.) Thus, an an attacker can circumvent the protections of

from a single server. The target starts and completes 15 timing exchanges with its server, averaging about one exchange per minute. We then attack, sending the target a spoofed zero-0rigin timestamp packet every second for ten seconds; these spoofed packets have T_3 as October 22, 1985 and T_2 as August 1, 2006. (This choice of T_2 sets $\delta \approx \psi \approx 0$, so we pass `TEST11`; see Section 2.) The target panics and restarts after the ninth spoofed packet. It then receives the tenth spoofed packet *before* it queries its server, and per the *reboot bug* in Appendix D, immediately shifts to 1985. (Our attack would still work even without this bug; the target would shift to 1985 after we sent it a few more packets.)

Eventually, the attacker decides to check if the attack has been successful by sending a mode 3 query to the target. By checking the transmit timestamp of the mode 4 response, the attacker realizes that the target is in 1985. Then, the attacker sets $T_2 = 0$ on his spoofed packets to maintain the target client in 1985. (This is necessary because the target client's T_4 is now October 22, 1985, so maintaining $\delta \approx 0$ so we pass `TEST11` requires $T_2 = 0$.) The attacker continues pelting the target with spoofed packets at a higher rate ($\approx$ 1 packet/second) than that of the legitimate server response packets ($\approx$ 1 packet/minute). The legitimate packets look like outliers (due, in part, to `TEST11`, Section 2) and the target sticks to the attacker's bogus time.

A.2 Interleaved pivot vulnerability.

We next consider a vulnerability introduced by NTP's interleaved mode, which is designed to allow for more accurate time synchronization.

What is interleaved mode? Recall that NTP uses timestamps on the packets to determine the offset θ between the client and the server. Because these timestamps must be written to the packet *before* the packet is sent out on the network, there is a delay between the time when the packet is 'formed' and the time when the packet is sent. This delay is supposed to introduce small errors in the offset. Interleaved mode eliminates this delay by spreading the computation of the offset (equation (2)) over *two* exchanges, rather than just one. In interleaved mode, hosts record timestamps for the moment that they actually transmit packet onto the network, and send them in the packet transmitted in the subsequent polling interval. Interleaved mode is not mentioned in RFC5905 [27], but is implemented in ntpd. Mills [28] indicates that interleaved mode is intended for use on top of the broadcast or symmetric modes only.

We will exploit the following issues: (1) Interleaved mode shares the same code path as the client/server code (Figure 11). (2) Interleaved mode *changes* the meanings of the values stored in the timestamp fields of an NTP packet. Importantly, the origin timestamp field of the mode 4 server response now contains T_4 from the previous exchange (rather than T_1 from the current exchange).

panic threshold by sending the client a timeshift that exceeds the panic threshold, causing the NTP daemon to quit. The OS subsequently automatically reboots the NTP daemon. Now, if the NTP daemon is running with the `-g` option, it will ignore the panic threshold because it has just rebooted.

Thus, the usual `TEST2` no longer works; instead, there is 'interleaved `TEST2`' comparing the packet's origin timestamp to the `rec` variable, which stores T_4 from the previous exchange. (Line 21 in Figure 11.) (3) A host *automatically switches* into interleaved mode when it detects that the host on the other side of the association is in interleaved mode. (Line 15 in Figure 11.)

Interleaved mode as a low rate DoS vector. The implementation of interleaved mode in ntpd introduces a low rate denial-of-service attack. The vulnerability is introduced in line 14 of Figure 11. Namely, if a server response packet fails the usual `TEST2`, the client subjects the packet to 'interleaved `TEST2`'. If the packet passes, the client sets the `interleave` variable and enters interleaved mode. Importantly, a client cannot escape from interleaved mode—there is no code path to clear the `interleave` variable.

Thus, an off-path attacker can inject a spoofed server response packet that passes 'interleaved `TEST2`' because its origin timestamp equals T_4 from the previous exchange. But how can the attacker learn T_4? It turns out that whenever an ntpd client updates its clock, it sets its reference time to be T_4 from the most recent exchange with the server to which it synchronized. This T_4 is sent out with every subsequent packet in the *reference timestamp* field.[17] Thus, to learn T_4, the off-path attacker first sends the client a regular mode 3 query, and learns the reference time from the client's response. If the target updated its clock in the previous exchange, the reference time will be T_4 from the previous exchange. The target will then react to the spoofed packet by switching into interleaved mode (Line 14 of Figure 11). All subsequent legitimate server responses are rejected because they fail 'interleaved `TEST2`'.

The attack leads to a DoS for *each one* of the target's servers. The attack works by repeating the process of (1) sending a timing query to the target to learn the IP of the server that the target synchronizes to, and its T_4 timestamp, and then (2) pivoting the target into interleaved mode for that server by sending a spoofed interleaved pivot packet. Thus, whenever the target synchronizes to a new server, the attacker will detect this (in step (1)) and DoS that new server as well (in step (2)). (This process is similar to the DoS by Spoofed Kiss-o'-Death attack from [21, Sec V.C].)

Timeshifting attacks. Section 4 shows how nptd's control query interface can be leveraged to turn the interleaved pivot vulnerability into a time-shifting attack (rather than just a DoS attack). Section 5.4 finds 700K vulnerable IPs.

B Flaws in Symmetric Mode

Some of the vulnerabilities in NTP's client/server mode (mode 3/4) follow because it shares the same code path as NTP's symmetric mode (mode 1/2). Therefore, we now consider the security of NTP's symmetric mode. We identify several flaws in its specification in RFC5905 (including several off-path denial-of-service (DoS) attacks on unauthenticated symmetric mode, and several replay

[17] Miroslav Lichvar noted that T_4 leaks in the reference timestamp.

attacks (*i.e.,* on-path DoS attacks) on authenticated symmetric mode), explain how these flaws harm client/server mode, and conclude with recommendations.

B.1 Background: Symmetric mode.

In symmetric mode, two peers Alice and Bob can give (or take) time to (or from) each other via either ephemeral *symmetric passive* (mode 2) or persistent *symmetric active* (mode 1) packets. The symmetric active/passive association is preconfigured and initiated at the 'active' peer (Alice), but not preconfigured at the 'passive' peer (Bob). Upon arrival of a persistent mode 1 NTP packet from Alice, Bob mobilizes a new ephemeral association if he does not have one already. Because this is a potential security risk—an arbitrary attacker ask Bob to become its symmetric peer and start offering time to Bob—ntpd requires symmetric passive associations to be cryptographically authenticated by default. Active/active symmetric associations are also possible, where both peers are preconfigured with persistent associations. In this case, authentication is *not* required by default.

Symmetric mode has two additional quirks. First, each peer uses its own polling algorithm to decide when to respond to its peer. As such, Bob will not immediately respond to Alice upon receipt of her packet. (This is in contrast to the client/server mode, where servers immediately respond to queries.) Second, both peers perform `TEST2` (and other tests) on the *same* volley of packets, and use the *same* packet timestamps to obtain timing samples.

B.2 Problems with bogus packets.

In both RFC5905 and ntpd, a host processes (mode 1 and 2) symmetric mode packets it receives using the same code used to process (mode 4) server response packets. Another look at this code in Appendix A of RFC5905 (Figure 5) shows that the `org` state variable is updated even when a received packet *fails* `TEST2`. ntpd prior to v4.2.8p4 does this as well (Lines 16 and 27 in Figure 11). But should a bogus packet really be allowed to update the client's state? We now explain why there is no easy answer to this question.

What if bogus packets do not update `org`? We first suppose that the `org` state variable is *not* updated upon receipt of a *bogus packet* (*i.e.,* a packet that fails `TEST2`). We show this leads to persistent failures in two cases:

1) Packet drop leads to persistent failure. In Figure 12 Alice's second packet to Bob is dropped. After Alice's packet is dropped, Bob's $\texttt{org}_b$ state variable still stores the (now stale) time T_1. Bob uses T_1 as the origin timestamp of the packet he now sends to Alice. Alice drops this packet because its origin timestamp T_1 does not match her $\texttt{xmt}_a = T_5$ variable. Now this 'bogus' packet also does *not* update Alice's $\texttt{org}_a$ variable. Next, Alice sends a new packet to Bob at time T_9, using the (now stale) value $\texttt{org}_a = T_3$ as the new packet's origin timestamp. Now Bob drops the packet, because its origin timestamp T_3 does not match $\texttt{xmt}_b = T_7$. This continues indefinitely, so all future packets fail `TEST2`.

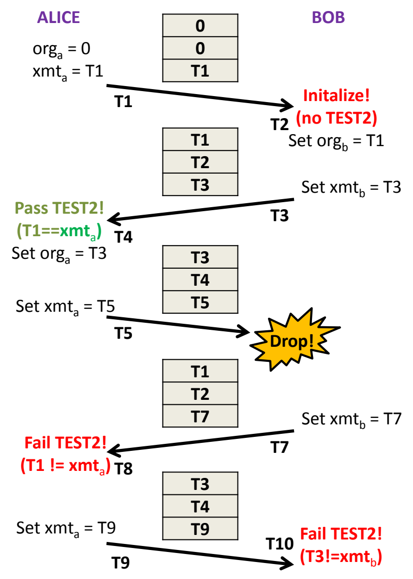

Fig. 12. Alice (left) exchanges symmetric mode packets with Bob (right). Each grey packet depicts the following fields (per Figure 4) in order: origin timestamp, receive timestamp, transmit timestamp. Alice's state variables $\texttt{org}_a$ and $\texttt{xmt}_a$ are shown on the left. Bob's state variables $\texttt{org}_b$ and $\texttt{xmt}_b$ are shown on the right. (Alice initializes the association by sending Bob an initialization packet, with origin and receive timestamps set to zero, and transmit timestamp set to Alice's sending time T_1. Alice writes T_1 to $\texttt{xmt}_a$. Bob receives the packet and copies the transmit timestamp T_1 from the packet to his $\texttt{org}_b$. When Bob's polling algorithm indicates he is ready to respond, he sends Alice a packet with origin timestamp T_1 copied from his $\texttt{org}_b$ and transmit timestamp T_3 equal to his time when he sent the packet. Bob then writes T_3 to his $\texttt{xmt}_b$. Upon receipt of Bob's packet, Alice performs TEST2, updates her state variables, computes offset, delay, *etc.*, and decides whether to update her clock. When her polling algorithm indicates that she is ready to respond, she constructs her next packet to Bob using her state variables in the same way Bob did.) In this Figure, Alice's second packet to Bob is dropped. If bogus packets (failing TEST2) do not update org, as shown here, then one dropped packet can cause persistent failure.

2) Unsynchronized poll leads to persistent failure. We saw a similar failure happen naturally during an authenticated active/active symmetric association between peers Alice and Bob both running ntpd v4.2.8p6. This version of ntdp does *not* update `org` upon receipt of a bogus packet (because of the return added on line 16 in Figure 11).

In Figure 13 Alice was a symmetric peer with Bob. Bob was a symmetric peer with Alice, and also a client to an external server. Alice had a clock synchronization event that caused her to set her polling interval to 64 seconds. Meanwhile, Bob's polling interval was 128 seconds. Next, Alice sent Bob a packet with the correct origin timestamp T_0 expected by Bob. Bob accepted this packet and cleared $\texttt{xmt}_b$ and updates his $\texttt{org}_b = T_a$. However, Bob did not yet respond, since his polling interval was longer than Alice's. In the meantime, Alice sent Bob another packet with this same origin timestamp T_0. (Alice sends the same T_0 because she has not yet received a new packet from Bob to cause her to update her $\texttt{org}_a$ variable.) This time Bob rejected the packet by `TEST2` because he had cleared $\texttt{xmt}_b$. When Bob was ready to respond to Alice, he sent a packet with origin timestamp T_a matching to that in the first packet sent by Alice. (This is because Bob did not update his $\texttt{org}_b$ variable from the second rejected packet.) But Bob's packet failed `TEST2` at Alice, because she was expecting origin timestamp T_c corresponding to the second packet. We are back in the persistent failure scenario of Figure 12.

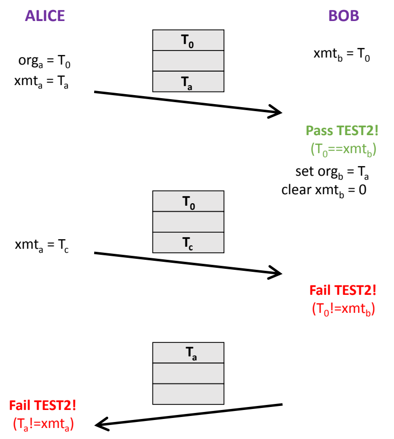

Fig. 13. Alice (left) exchanges symmetric mode packets with Bob (right). Alice sends two consecutive packets to Bob due to unsynchronized polling intervals. The first packet passes `TEST2`, but all subsequent packets fail `TEST2` on both peers, leading to persistent failure.

What if bogus packets do update `org`? The reader might now conclude that bogus packets *should* update `org`, as is required by Appendix A of RFC5905. However, this leads to two denial-of-service attacks:

1) On-path denial-of-service for authenticated symmetric mode. Suppose that `org` can be updated by bogus packets that pass cryptographic validation (of the MAC) but fail `TEST2`. Consider an on-path attacker (who does not have the ability to drop/modify/delay packets) who attacks an *authenticated* symmetric association. (Note that symmetric active/passive associations are authenticated by default.) We show that this on-path attacker can parlay his ability to replay packets into the ability to (effectively) drop packets.

To do this, the on-path attacker replays any stale packet from Alice to Bob (1) after Alice sends Bob a legitimate packet but (2) before Bob sends his response

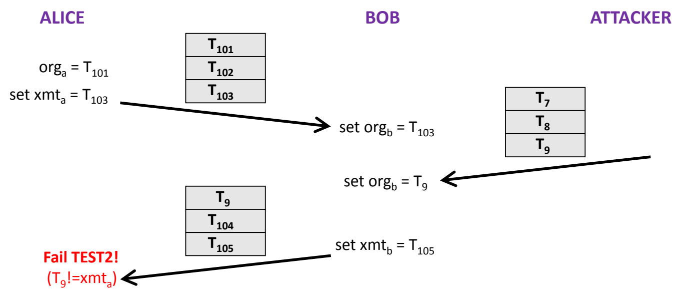

Fig. 14. Alice (left) exchanges symmetric mode packets with Bob (center). Attacker (right) is *on-path* for authenticated NTP and *off-path* for unauthenticated NTP. For the on-path authenticated DoS attack, the attacker's packet (timestamps T_7, T_8, T_9) is a replay of a stale packet sent from Alice to Bob. For the off-path unauthenticated DoS attack, the attacker's packet is spoofed.

as per Figure 14. If these polling intervals are not synchronized, the attacker has plenty of time (*i.e.,* seconds or minutes) to perform this replay. The stale replayed packet overwrites Bob's $\texttt{org}_b$ variable to T_9. Thus, Bob responds to Alice with a packet whose origin timestamp is equal to the (stale) transmit timestamp T_9 from the stale replayed packet. This stale origin timestamp T_9 fails `TEST2` at Alice. The attacker can repeat this replay each time Alice sends Bob a packet, thus preventing Alice from ever synchronizing to Bob.

Note also this attacker need not be 'on-path' forever. Indeed, once the attacker gets his hands on a single stale packet sent from Alice to Bob, he can move off-path, and keep launching this attack forever by replaying this stale packet.

2) Off-path denial-of-service for unauthenticated symmetric mode. Suppose `org` can be updated by bogus packets that fail `TEST2`. We show how an off-path attacker can launch an identical attack on *unauthenticated* symmetric mode by spoofing (rather than replaying) a packet from Alice. This is a serious threat, since active/active symmetric associations are not cryptographically-authenticated by default.

We performed this attack on two ntpd v4.2.8p2 hosts Alice and Bob. (We use v4.2.8p2 because this implementation lets bogus packets update `org`.) Both Alice and Bob are preconfigured to be each other's symmetric active peer. Additionally, Bob is also preconfigured in client/server mode with four other servers. Upon restarting ntpd on both hosts, Bob gets synchronized to one of his servers in the very first exchange (per the reboot bug, see Appendix D). Alice sends symmetric active mode packets to Bob and gets back symmetric active response packets from Bob. After four exchanges with Bob, Alice synchronizes to Bob and indicates this by putting Bob's IP address in the *reference ID* of her fifth packet. After two more exchanges, the off-path attack begins.

In symmetric mode, the time between a packet and its response is often up to several seconds (62 seconds in this experiment), giving our off-path attacker plenty of time to inject packets. So the attacker sends Bob a symmetric mode packet spoofed to look like it came from Alice; this query is sent after Alice sends her legitimate query to Bob, and before Bob sends his reply (per Figure 14). Bob updates $\texttt{org}_b = T_9$ from attacker's bogus packet. Now Bob sends the response to Alice with origin timestamp T_9 corresponding to $\texttt{org}_b$. This packet fails TEST2 at Alice. The attacker continues to inject spoofed packets to Bob for the next 16 exchanges between Alice and Bob. Bob's responses fail TEST2 at Alice and so Alice never updates her clock.

Summary. Thus, we are between a rock and a hard place. Should bogus packets update org or not? Our recommendations are in Appendix B.4.

B.3 Problems with initialization.

Consider what happens if Alice reboots and sends Bob a packet initializing their association. Alice has no timing information, so this 'initialization packet' has $T_1 = T_2 = 0$ (as in the first packet in Figure 12). If Bob did not reboot, he has xmt != 0. Now, if Bob performed TEST2 on the initialization packet, it would be dropped (because T_1 != xmt). Also, it would be dropped if Bob performed TEST3 (because $T_1 = T_2 = 0$). Thus, if the protocol is to tolerate a reboot, initialization packets cannot be subject to TEST2 or TEST3.

Denial-of-service via initialization packets. We use the fact that TEST2 cannot be performed on initialization packets to perform DoS attacks identical to those in Appendix B.2. We can perform on-path DoS attacks on authenticated symmetric associations by replaying initialization packets (instead of replaying stale packets). Off-path DoS attacks on unauthenticated symmetric mode can also be accomplished by spoofing initialization packets (rather than spoofing arbitrary packets); notice that spoofing initialization packets is trivial because they do not contain any unpredictable information. Importantly, both of these DoS attacks exist *regardless of* whether bogus packets update org or not (Appendix B.2).

Impact on client/server mode. The initialization of symmetric mode requires that TEST2 and TEST3 are not performed on a received packet with a zero-origin timestamp. However, this is at odds with the security of client/server mode. Unfortunately, however, client/server and symmetric modes share the same code path. ntpd deals with symmetric mode initialization using Lines 10-11 in Figure 11, which clears xmt and skips TEST2 if a received packet has a zero-origin timestamp. These lines of code, however, create the zero-0rigin timestamp vulnerability in client/server mode (Appendix A.1). Meanwhile, RFC5905 Appendix A deals with this by not performing TEST3 (Figure 5). However, because TEST3 is not performed, the xmt variable cannot be cleared, creating the query-replay vulnerability (Section 3.2).

B.4 Symmetric Mode: Choose your poison!

Many problems in symmetric mode occur because both peers update their state variables (org, xmt) and collect timing samples $(\theta_i, \delta_i, \psi_i)$ from the *same* volley of packets. Per the discussion in Appendix B.2, we cannot see how to fix this while maintaining a single volley of packets between peers. One drastic suggestion is to require *two* distinct volleys, where each peer is a server in one volley, and is a client in the other (using one of the protocols described in Section 6.1). However, this is not backwards compatible, as both peers involved in association must simultaneously make this change. Thus, an (unsatisfying) band-aid solution could involve:

1. To prevent the persistent failure problem of Appendix B.2, allow packets failing `TEST2` to update `org`. However, this enables off-path DoS attacks.
2. To prevent off-path DoS attacks, we suggest mandatory cryptographic authentication in symmetric mode (for both active/active and active/passive).
3. Even so, symmetric peers that use cryptographic authentication are still vulnerable to DoS attacks, so we also suggest monitoring to detect excessive number of bogus packets (Appendix B.2).
4. Monitoring should also be used to detect excessive number of initialization packets, since these also lead to DoS (Appendix B.3).
5. Finally, symmetric peers should ensure that they run `TEST2` against an origin timestamp that contains 32 bits of randomness. This can be done with a receive function as in Figure 6 and a sending function in Figure 7.[18]

C On-path query replay attack

Replays of the client's query are a problem because they harm the accuracy of time synchronization. We demonstrate this with an on-path query replay attack on a target host in our lab. We were able to degrade the accuracy of the target's time synchronization from 4×10^{-5} seconds (on average) to 2.7 seconds (on average). We show the client's offset (*i.e.,* the distance between the client's clock and the server's clock per equation (2)) under normal and attack conditions in Figure 15; the attacked client's accuracy is 5 orders of magnitude worse.

Experiment. We modified the source code for ntpd v4.2.8p2 to make it vulnerable to client query replays. Specifically, we deleted the line of code that cleared `xmt` when a response passed `TEST2`. We then preconfigured our modified ntpd client with one server. Every time the client sent a query to its server, our on-path attacker captured the query, and replayed it to the server once per second, until the client sent a new query. We repeated this for every query sent by the client. The resulting offset in Figure 15 was computed per

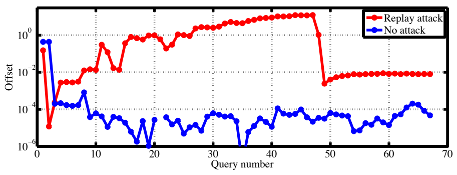

Fig. 15. Query replay attack on modified version of ntpd v4.2.8p2. Time synchronization on the attacked client degrades by 10^5.

[18] Because both peers update their state variables and collect timing samples from the same volley of packets, symmetric mode must preserve the semantics of the origin timestamp. Thus, in symmetric mode we cannot replace the origin timestamp with a random 64-bit nonce per Figure 8.

equation (2) with T_1, T_2, T_3 taken from the NTP packet timestamps on server responses sent in response to real client queries (not replayed queries) and T_4 taken from the response packet's arrival time at the client. We repeat this experiment on the same client and server machine but without a replay attack.

Why does accuracy degrade? This follows because the replayed queries cause the server to respond with a stale origin timestamp T_1. Suppose that t seconds elapse since the client's most recent query. If the attacker now replays the client's query, the packet timestamps in the server's response will be such that $T_2 - T_1 \approx t$ seconds and $T_4 \approx T_3$ seconds, resulting in a timing sample with offset $\theta \approx \frac{t}{2}$ seconds per equation (2). As t grows during the polling interval, the offset in the timing sample grows as well. Thus, when the client uses these sampled offsets to set its clock, it miscalculates the discrepancy between its local clock and that of the server, resulting in the inaccuracies in Figure 15. Thus, this query replay attack has similar effect to a delay attack [30].

D Reboot bug

Our experiments show that, upon reboot, an ntpd v4.2.8p6 client updates its local clock from the very first response packet it receives from *any* of its preconfigured servers. This is CVE-2016-7433. We now explain why this is a security vulnerability.

What does the RFC say? When describing the algorithm used for clock updates, the pseudocode in Appendix A.5.2 of RFC5905 has a comment that states "select the best from the latest eight delay/offset samples". Also, a client configured with multiple servers is supposed to choose the 'best' server from which it will take time. Section 5 says: "The selection algorithm uses Byzantine fault detection principles to discard the presumably incorrect candidates called "falsetickers" from the incident population, leaving only good candidates called "truechimers"." We argue that this bug disables Byzantine fault tolerance upon reboot.

Experiment. We set up an ntpd v4.2.8p6 client preconfigured with five pool servers. Upon reboot, the client sends server Alice a mode 3 query with *reference id* 'INITALIZATION' (indicating that it is unsynchronized) and *reference time* 'NONE' (expected behavior upon reboot). Another such query is made to Bob. Bob's response arrives first, followed by the response from Alice. Next the client sends a query to server Carol. The *reference id* field in this new query is Bob's IP, and the *reference time* is set to a time *before* the response was received from Alice. We therefore see that the client updates his clock upon receipt of his first response packet (from Bob), without considering the contributions of servers Alice, Carol, Dave and Frank.

Implications. Thus, on reboot, Byzantine fault tolerance is disabled, and the client is at risk of taking time from bad timekeepers. This issue becomes even more serious when the panic threshold is disabled upon reboot when a client is configured with `-g` option. (This is the default on many OSes, see footnote **??**.)

Thus, if a bad timekeeper's response arrives first, a `-g` client will immediately accept huge, potentially bogus, update to its clock.

Worse yet, an off-path attacker can exploit this, along with other bugs, in order to perform a low-rate time-shifting attack. The attacker first learns the IP of one of the target's preconfigured servers, using the trick per footnote 6. Then, the off-path attacker sends some 'packet-of-death' that crashes ntpd.[19] If the OS reboots ntpd, then the target restarts with the panic threshold disabled. The attacker now injects a single spoofed server response with (1) zero origin timestamp (per Appendix A.1), (2) the legitimate server as the source IP, and (3) some huge (incorrect) time offset. The client accepts the response even before it queries its legitimate servers, and adjusts its clock to the attacker's bogus time. The low rate of this attack—it requires only three packets—also means it could be sprayed across the Internet.

Recommendation. An NTP client should be compliant to the RFC specifications even upon reboot, and adjust its clock only after multiple successful exchanges with each of its timeservers.

Where is the bug introduced? The bug is introduced in ntpd v4.2.7p385 (released August 18, 2013) and exists in all the following versions, upto and including ntpd v4.2.8p7 (released April 26, 2016). The definition for *Root distance* (λ) in the variable *dtemp* in file `ntp_proto.c` was changed between ntpd v4.2.7p384 (Figure 17) and ntpd v4.2.7p385 (Figure 16). This change (Lines 2966-2967 in Figure 16) introduced the bug. However, this change violates compliance with the definition of *Root distance* in RFC5905 which defines it as in Figure 17.

Tested Versions. ntpd v4.2.7p384, ntpd v4.2.7p385, ntpd v4.2.8p6, ntpd v4.2.8p7, ntpd v4.2.8p8. This part of code remains the same in all the versions beginning ntpd v4.2.87p385.

Patch: Replacing code in Figure 16 with that in Figure 17 mitigates the bug. We successfully patched ntpd v4.2.8p7 (lines 3447-3453). To confirm, we ran the experiment with the patched version of ntpd v4.2.8p7 with the same setup as above. The test client updates its local clock after obtaining four timing samples from its servers.

```
2965   dtemp = (peer->delay + peer->rootdelay) / 2
2966           + LOGTOD(peer->precision)
2967           + LOGTOD(sys_precision)
2968           + clock_phi * (current_time - peer->update)
2969           + peer->rootdisp
2970           + peer->jitter;
```

Fig. 16. Lines 2965-2970 in ntpd v4.2.7p385

[19] For instance, CVE-2016-7434 or CVE-2016-9311.

```
2933   dtemp = (peer->delay + peer->rootdelay) / 2 + peer->disp
2934           + peer->rootdisp + clock_phi * (current_time - peer->update)
2935           + peer->jitter;
```

Fig. 17. Lines 2933-2935 in ntpd v4.2.7p384

E Disclosure and Subsequent Developments

This research was done against ntpd v4.2.8p6, which was the latest version of ntpd until April 25, 2016. Since that date, three new versions of ntpd have been released: ntpd v4.2.8p7 (April 26, 2016), ntpd v4.2.8p8 (June 2, 2016), and ntpd v4.1.8p9 (November 22, 2016). We summarize our disclosure timeline and the impact of our research results on new releases of ntpd as follows:

Report. This report was first disclosed on June 7, 2016 and Section 5 was revised on July 29, 2016. The report was last edited, for clarity and style, on February 20, 2017.

Zero-Origin timestamp vulnerability (CVE-2015-8138, CVE-2016-7431, Appendix A.1). This vulnerability was disclosed in October 2015 (prior to ntpd v4.2.8p4) but unfortunately still existed in v4.2.8p8. (This is because Lines 10-11 of Figure 11 were still present in ntpd v4.2.8p8, likely in order to process initialization packets in symmetric mode, see Appendix B.3.) The vulnerability has been fixed in ntpd v4.2.8p9.

Interleaved pivot vulnerability (CVE-2016-1548, Appendix A.2). Following disclosure of this vulnerability in November 2015[20], ntpd v4.2.8p7 was patched so that clients do not automatically switch into interleaved mode by default. Now, clients do this only if the option '`xleave`' is set with a `peer`, `server` or `broadcast` configuration command.

Leaky control queries (CVE-2015-8139, Section 4). This vulnerability was first disclosed in October 2015, but ntpd v4.2.8p9 still accepts control queries from arbitrary IPs by default. Users must configure the `noquery` option to change this default. The leaky control queries we describe are also mentioned in a new Internet draft [26]. We have worked with the authors of [26] to add a security considerations to this document.

Origin timestamp randomization (Section 6). Cryptographic randomization of the origin timestamp has not yet been incorporated into ntpd.

Bogus packets in symmetric mode (Appendix B.2). ntpd v4.2.8p7, ntpd v4.2.8p8 and v4.2.8p9 allow bogus packets (that fail `TEST2`) to update the `org` state variables.[21] Thus, the 'transient failure to persistent failure' from Appendix B.2 is no longer present, but the two denial of service vulnerabilities in Appendix B.2 are present. This issue was disclosed on June 7, 2016.

[20] Also the concurrent disclosure by Miroslav Lichvar.

[21] This change was probably done in response to NTP Bug2952 reported by Michael Tatarinov and made public on April 26, 2016 (concurrently with our work). The bug report states only that "symmetric active/passive mode is broken" [42].

DoS via initialization packets in symmetric mode (Appendix B.3). This was first disclosed on June 7, 2016. These flaws were still present in v4.2.8p9.

Reboot bug (CVE-2016-7433, Appendix D). We first disclosed this issue in August 2015, and provided a full analysis on June 7, 2016. This bug was still present in v4.2.8p8 and fixed in v4.2.8p9.

F Security analysis

In this appendix, we provide our formal security model and prove that the protocols in Appendix 6.1 are secure against off-path attackers when NTP is unauthenticated and against on-path attackers when NTP is authenticated.

F.1 Model

Our model focuses on the description of an honest party called the *network* $\mathcal{N}$ that delivers packets and orchestrates the execution of several NTP exchanges akin to the environment in a UC protocol [8].

Parties. We suppose there are ℓ honest parties $\mathcal{P}_1, \ldots, \mathcal{P}_\ell$, where $\mathcal{P}_i$ denotes the IP address of the i^{th} party, who collectively perform many pairwise client/server exchanges. A single party $\mathcal{P}_i$ may act in both the client and server roles in different exchanges. There is also single attacker $\mathcal{A} = \mathcal{P}_0$ who may also have honest client/server exchanges, but has other goals and powers as well. Parties send packets through a *network* $\mathcal{N}$.

Packets. We model a packet as a tuple of the form (h, m, t), where $h = (\mathsf{IP\_src}, \mathsf{IP\_dst})$ contains the source and destination IPs, t is a MAC tag optionally appended to the packet, and m contains the remaining fields of the packet that are authenticated by the MAC tag.

Packet delivery. The *network* $\mathcal{N}$ maintains a counter step to orchestrate the flow of communication. Informally, one may think of step as the wall-clock time from $\mathcal{N}$'s point of view. During each step of the step counter, each honest party may receive, process, or transmit one packet.

We require that honest parties have sufficient time to process every packet received; put another way, attacks that flood an honest party with packets in order to deny service are out of scope. Formally, we model this property by (1) restricting $\mathcal{N}$ to deliver at most one packet per step of the counter to each party, and (2) incrementing the counter in integer multiples of L/R, where R denotes an upper bound on the bandwidth (bps) of honest parties when ingesting NTP packets of length L bits. Here, $L = 720$ bits for unauthenticated NTP packets and $L = 720 + 2n$ bits for NTP packets authenticated with a MAC of length $2n$.

The network imposes a constant latency ϱ on packet delivery. Specifically, if $\mathcal{N}$ receives a packet from an honest party when the counter is step, $\mathcal{N}$ holds the packet in a queue and assigns a value $\mathsf{deliver} = \mathsf{step} + \varrho$ to the packet. When $\mathsf{step} == \mathsf{deliver}$, the packet is transmitted. We stress that the attacker $\mathcal{A}$ does *not* have the power to modify, delay, or drop packets between honest parties.

Race conditions. Unlike the honest parties, attacker $\mathcal{A}$ may specify the $\mathsf{deliver}$ value for all packets she sends. However, as per constraint (1) above, this value must be distinct from the $\mathsf{deliver}$ values of all other packets in $\mathcal{N}$'s queue destined for $\mathsf{IP\_dst}$.

This power allows $\mathcal{A}$ to win all race conditions.[22] As such, our model allows attakcer $\mathcal{A}$ to send an honest party up to $2\varrho\frac{R}{L}$ packets in the duration of an NTP exchange, since $\mathcal{A}$ can request the delivery of one packet for each step of the counter, while the two messages in an NTP exchange are each subject to a longer delay ϱ.

Additionally, this power also encapsulates the real-world uncertainty in packet delivery. So far, our model assumes that all packets encounter a constant delay. In reality, we often have at most a rough upper-bound on network latency, and we want for NTP's security guarantee to hold for any distribution of packet latency times that fit within this bound. Rather than formalizing a network latency distribution within our model, we opt for the simpler approach of letting attacker $\mathcal{A}$ "speed up" packet arrivals using its power to determine $\mathsf{deliver}$ for its packets.

Transcripts. A *transcript* is a list of NTP client/server exchanges, formally written as a set of tuples

$$(\mathsf{start}, i, t_c, j, t_s)$$

each indicating that an exchange between client $\mathcal{P}_i$ with local time t_c and server $\mathcal{P}_j$ with local time t_s starts at step start of the step counter. We stress that $\mathcal{A} = \mathcal{P}_0$ may legitimately engage in NTP exchanges in the transcript specification.

Network $\mathcal{N}$ enforces execution of the transcript by informing parties that they should begin a client/server exchange. When $\mathcal{N}$'s counter is $\mathsf{step} == \mathsf{start}$ for tuple $(\mathsf{start}, i, t_c, j, t_s)$ in the transcript ts, $\mathcal{N}$ sends a 'go message' (client, i, t_c) to $\mathcal{P}_i$, after which an honest party $\mathcal{P}_i$ immediately sets her local clock to t_c, runs the protocol in Figure 7 or Figure 8 resulting in a mode 3 query packet $((\mathcal{P}_i, \mathcal{P}_j), m, t)$ to $\mathcal{P}_j$ through $\mathcal{N}$. After a delay of ϱ, $\mathcal{N}$ sends a 'go message' (server, i, t_s) to the $\mathcal{P}_j$ and also delivers the $\mathcal{P}_i$'s query packet, and honest $\mathcal{P}_j$ responds assuming that her local clock is set to time t_s.

The transcript must be consistent with our "no flooding" rule that limits each party to receiving 1 packet (plus perhaps a 'go message') per step. As a consequence, two exchanges involving the same client cannot have the same start counter. Additionally, a party cannot simultaneously be a server at counter start and a client at counter $\mathsf{start} + \varrho$.

Interacting with the network. The network $\mathcal{N}$ starts by receiving ts and then choosing and dispersing secret keys for each pair of parties $\{\mathsf{sk}_{i,j} : i, j \in \{0, 1, \ldots, \ell\}\}$. The honest parties $\mathcal{P}_i$ receive these keys and initialize their $\mathtt{xmt}_j$ state variables for every other party $j \neq i$. The game then begins with $\mathcal{N}$ in control and the counter step initialized to 0.

[22] We remark that this capability is unrealistically powerful for an off-path attacker, who cannot observe honest packet transmissions.

If $\mathcal{N}$'s counter step equals either (1) the start value of a tuple in the transcript ts or (2) the $\mathsf{deliver}$ value of a packet in its queue, then $\mathcal{N}$ delivers the appropriate packet or 'go message' and cedes control to the honest party. The honest party starts computing when it receives the packet or 'go message'.

Once an honest party finishes its computation and possibly transmits a new packet $((\mathsf{IP\_src}, \mathsf{IP\_dst}), m, t)$ to $\mathcal{N}$, the honest party cedes control back to $\mathcal{N}$. Next, the network $\mathcal{N}$ instantly reveals $[(\mathsf{IP\_src}, \mathsf{IP\_dst}), \mathcal{L}(m, t)]$ to the attacker, where $\mathcal{L}$ is a *leakage function.* $\mathcal{N}$ then cedes control to $\mathcal{A}$, who may perform arbitrary computations and optionally transmit a packet of its own. When $\mathcal{N}$ regains control, it increments step and repeats the process. This model implicitly forbids $\mathcal{A}$ from dropping, modifying, or further delaying packets; instead, every packet is delivered intact to $\mathsf{IP\_dst}$ after delay ϱ.

Leakage. The *leakage function* $\mathcal{L}$ models the information available to an on-path or off-path attacker. Specifically, $\mathcal{L}$ equals the identity function for an on-path attacker (*i.e.,* m and t are revealed perfectly) and the zero function for an off-path attacker (*i.e.,* m and t are perfectly hidden).

Spoofing. Network $\mathcal{N}$ never validates $\mathsf{IP\_src}$ in a transmitted packet. This allows attacker $\mathcal{A}$ to send packets with a spoofed source IP $\mathsf{IP\_src}$. Meanwhile, honest parties always use their true $\mathsf{IP\_src}$. Additionally, if $\mathcal{A}$ spoofs a query packet on behalf of a client $\mathcal{P}_i$, we observe that $\mathcal{N}$ lacks a timestamp t_s^* to deliver to the honest server $\mathcal{P}_j$ along with the query packet. We choose t_s^* as follows: if $\mathcal{A}$'s spoofed packet occurs during an honest NTP transaction between parties i and j, then $\mathcal{N}$ sends the same timestamp that the honest transaction uses; otherwise, $\mathcal{A}$ may choose t_s^* arbitrarily and inform $\mathcal{N}$ of its choice.

Soundness guarantee. Without an attacker, the results of honest parties' NTP exchanges are completely defined by the transcript. Formally, clients update their local *state* which includes the set of servers they are willing to query, the state variables (*i.e.,* $\mathtt{xmt}_j$ and $\mathtt{org}_j$) used in exchanges with each server $\mathcal{P}_j$, and the resulting set of validated timing samples (including delay δ, offset θ, jitter ψ, dispersion ϵ per the equations in Section 2). $k\text{-}\mathsf{state}_i(\mathsf{ts}, \mathsf{step})$ is the state of party $\mathcal{P}_i$ during an execution of the transcript ts while $\mathcal{N}$'s counter is step, and contains the results of the k most recent exchanges with each server.

$\mathcal{A}$'s objective is to tamper with k consecutive timing samples that some honest client $\mathcal{P}_i$ stores in its state corresponding to interactions with a single server $\mathcal{P}_j$.[23] Hence, we let $k\text{-}\mathsf{state}_i^{\mathcal{A}}(\mathsf{ts}, \mathsf{step})$ denote the state of party $\mathcal{P}_i$ during a game where the attacker $\mathcal{A}$ is present. Of course, if $\mathcal{P}_i$ voluntarily chooses to query $\mathcal{A}$ as its server, then $\mathcal{A}$ can significantly influence $\mathcal{P}_i$'s state. The soundness guarantee effectively states that $\mathcal{A}$ can do no more than this.

[23] Note that the definition 'NTP exchanges should not fail' does not hold because exchanges may fail even without an attacker. As one example, consider a client who initiates two exchanges with the same server in rapid succession, *i.e.,* the client's second query is sent before she receives a response to the first query. Then TEST2 will fail for the server's first response.

However, there is one type of modification that we cannot hope to rule out. Consider the effect of $\mathcal{A}$ "preplaying" honest packets: that is, submitting a packet that is identical to one in $\mathcal{N}$'s queue but with an earlier arrival time. This action is very likely to affect the state of honest parties, albeit in a bounded manner. It may reduce delay measurements δ from their upper bound of 2ϱ, but never increase them. Similarly, each offset θ may increase or decrease by at most ϱ. Finally, jitter ψ may be altered slightly, likely by far less than the bound accepted by `TEST11`. Due to their limited, unavoidable effects, we consciously opt to ignore preplay attacks in the following soundness definition to simplify our discussion.

Definition F1 (Soundness)**.** NTP is (k, ϵ)-*sound* on transcript ts if for all *resource-bounded* attackers $\mathcal{A}$ who never preplay packets from honest parties, and for all parties $\mathcal{P}_i$ who do not query $\mathcal{A}$ as an NTP server,

$$\Pr[\exists\, \mathsf{step} \text{ s.t. } k\text{-}\mathsf{state}_i^{\mathcal{A}}(\mathsf{ts}, \mathsf{step}) \neq k\text{-}\mathsf{state}_i(\mathsf{ts}, \mathsf{step})] < \epsilon.$$

This inequality must hold for *all* k components of the state. The probability is taken over the randomness of all parties and $\mathcal{N}$'s choice of shared secret keys.

F.2 Soundness against off-path attackers.

We now prove Theorem 61 of Appendix 6.3, which states that the protocols in Appendix 6.1 are sound against an off-path attacker $\mathit{off}\mathcal{A}$. Theorem 61 follows largely from the entropy E present in the origin timestamp. We do not require NTP packets to be authenticated.

The theorem holds as long as randomness is produced from a cryptographically-strong random number generator (RNG), and that, upon reboot, honest parties initialize their $\mathtt{xmt}_j$ variables for each server to a 64-bit number generated by their RNG.

Let $\mathit{off}\mathcal{A}$ be any off-path attacker, and let ts be any transcript that involves ℓ honest parties, a maximum of τ exchanges involving any single client-server pair, and a maximum of s trusted servers per client.

Let i^* be any client who does not query $\mathit{off}\mathcal{A}$ as server. We say that the protocol described in Figure 6, 7 randomizes the sub-second granularity of the expected origin timestamp, while the protocol in Figure 6, 8 randomizes the entire expected origin timestamp.

We use a sequence of games to prove that $\mathit{off}\mathcal{A}$ tampers the state of $\mathcal{P}_{i^*}$ with probability at most $\epsilon_{\mathit{off}\mathcal{A}}$.

Game G_0. This is the real interaction of $\mathit{off}\mathcal{A}$ with the honest parties $\mathcal{P}_1, \ldots, \mathcal{P}_l$ and the network $\mathcal{N}$. For ease of notation, we denote the probability that $\mathit{off}\mathcal{A}$ breaks the soundness of game G_0 by $\Pr^0_{\mathit{off}\mathcal{A}}$.

Game G_1. This game is identical to G_0, except that $\mathcal{P}_{i^*}$'s pseudorandom number generator is replaced with a truly random number generator. By definition, the probability that anybody (in particular $\mathit{off}\mathcal{A}$) notices this change is at most Adv(RNG). Hence, $\Pr^0_{\mathit{off}\mathcal{A}} - \Pr^1_{\mathit{off}\mathcal{A}} \leq \text{Adv(RNG)}$.

Game G_2. This game is identical to G_1, except that we abort the execution if *off$\mathcal{A}$* sends a spoofed packet (*i.e.,* one for which *off$\mathcal{A}$* claims an $\mathsf{IP\_src}$ different than her own) involving client $\mathcal{P}_{i^*}$ and some server $\mathcal{P}_j$ such that the spoofed packet's origin timestamp matches $\mathcal{P}_{i^*}$'s state variable $\mathtt{xmt}_j$. Importantly, *off$\mathcal{A}$* has no chance of winning game G_2 (that is, $\Pr^2_{\mathit{off}\mathcal{A}} = 0$) because its spoofed packets always fail $\mathsf{TEST2}$. In order to demonstrate that any client $\mathcal{P}_{i^*}$ distrusting *off$\mathcal{A}$* properly refuses all of the attacker's spoofed packets while also accepting all of the honest servers' packets (*i.e.,* computes the desired value $k\text{-}\mathsf{state}_i(\mathsf{ts}, \mathsf{step})$ at all steps), it only remains to prove that the probabilities of winning G_1 and G_2 are close.

There are two conditions that cause Game G_2's abort condition to trigger:

– Client $\mathcal{P}_{i^*}$ is engaged in an NTP exchange with server $\mathcal{P}_j$ at the moment the spoofed packet is received, and the spoofed packet's origin timestamp matches that of the transmit timestamp in honest client's query.
– Client $\mathcal{P}_{i^*}$ is *not* engaged in an NTP exchange with server $\mathcal{P}_j$ at the moment the spoofed packet is received, and the spoofed packet's origin timestamp matches the client's randomly-chosen $\mathtt{xmt}_j$ value.[24]

In the first case, we know that the client's choice of the origin timestamp expected in the mode 4 response packet (*i.e.,* `pkt.T3` in Figure 7) ensures that $\mathtt{xmt}_j$ has $E = 32$ bits of entropy if the sub-second granularity of the timestamp comes from $\mathcal{P}_i$'s RNG or $E = 64$ bits of entropy if the entire origin timestamp is randomly chosen (*i.e.,* `pkt.T3` in Figure 8). When targeting a particular server $\mathcal{P}_j$, *off$\mathcal{A}$* can send $\frac{R\varrho}{360}$ packets to each honest party during an NTP exchange between the target client $\mathcal{P}_{i^*}$ and the server $\mathcal{P}_j$, each of which influences $\mathcal{P}_{i^*}$'s state with probability at most 2^{-E} where E is the number of bits of entropy in the origin timestamp. Hence, *off$\mathcal{A}$*'s ability to impact $\mathcal{P}_{i^*}$'s state during the NTP exchange is at most $Q_d = 2^{-E}\frac{R\varrho}{360}$.

In the second case, *off$\mathcal{A}$* can send $T = 2^p R/720$ packets to each party in the interval between two successive exchanges (where p corresponds to the polling interval). Each packet succeeds in altering $\mathcal{P}_{i^*}$'s state with probability $Q_b = 2^{-64}$ because $\mathtt{xmt}_j$ has 64 bits of entropy.

Finally, Lemma F1 below states that *off$\mathcal{A}$* can influence the state of k consecutive exchanges between client $\mathcal{P}_{i^*}$ and server $\mathcal{P}_j$ with probability at most $(k+1) \cdot (kQ)^k$, where $Q = \max\{Q_d, TQ_b\}$. Additionally, there are τ possible locations for this run of k successes to start, and s possible servers whose state may be attacked. In total, we find that:

$$\Pr^1_{\mathit{off}\mathcal{A}} - \Pr^2_{\mathit{off}\mathcal{A}} \leq (k+1)s\tau \cdot (kQ)^k$$

[24] Recall that $\mathcal{A}$ may choose the server's response time t_s^* arbitrarily in this case, which would have immense power if $\mathcal{A}$ could get the spoofed client to accept the response packet.

In practice, we claim that $Q_d > TQ_b$ if entropy $E = 32$:

$$2^{-32} \cdot \frac{R\varrho}{360} > 2^{-64} \cdot 2^p \frac{R}{720}$$
$$2^{33}\varrho > 2^p$$

With the maximum poll value $p = 17$ permitted by NTP [27], this reduces to the claim that $\varrho > 2^{-16} \approx 10^{-5}$ seconds, which is the time required for light to travel about 3 miles. So, our inequality is reasonable unless the client and server are physically co-located but still using a large polling value. Conversely, we claim $TQ_b > Q_d$ if entropy $E = 64$: this claim reduces to the statement that $2^p > 2\varrho$, which holds since RFC5905 constrains poll $p \geq 4$ while network delays δ do not exceed 16 seconds in practice.

All that remains is to prove the following combinatorial statement relating the probabilities of success during and between exchanges.

Lemma F1. *Let Q_d denote the probability that an attacker $\mathcal{A}$ successfully impacts the state of client $\mathcal{P}_{i^*}$ during an NTP exchange, Q_b denote the probability that each packet by $\mathcal{A}$ in between NTP exchanges impacts $\mathcal{P}_{i^*}$'s state, and let $T = 2^p \cdot \frac{R}{720}$ denote the number of packets that $\mathcal{A}$ may send to each party in between NTP exchanges. Then, the probability that $\mathcal{A}$ impacts k state observations in a row, beginning with a specified exchange, is at most $(k+1) \cdot (kQ)^k$, where $Q = \max\{Q_d, TQ_b\}$.*

Proof. $\mathcal{A}$ may compromise a total of k states either during or between exchanges. Let $c \in \{0, 1, \ldots, k\}$ denote the number of consecutive NTP exchanges (with a specified starting point) that $\mathcal{A}$ plans to compromise; clearly, she may do so with probability Q_d^c. Additionally, $\mathcal{A}$ must also inject a total of $k - c$ state measurements over the course of $c + 1$ intervals between these NTP exchanges. Here, each packet is an independent Bernoulli random variable that successfully impact's the client's state with success probability Q_b. The total number of between-exchange successes (i.e., the sum of the $(c+1)T$ Bernoullis) is distributed as a binomial random variable, hence the probability of $k - c$ total successes is at most $\binom{(c+1)T}{k-c} \cdot Q_b^{k-c}$.

In total, $\mathcal{A}$ succeeds at compromising c NTP exchanges and successfully injecting state $k - c$ times in between these exchanges with probability at most

$$\binom{(c+1)T}{k-c} Q_d^c Q_b^{k-c} \leq (kQ)^k,$$

where the inequality follows from the bound $\binom{x}{y} \leq x^y$. The lemma then follows by summing the probabilities of success for the $k + 1$ choices of c.

□

F.3 Soundness against on-path attackers.

We now prove Theorem 62 of Appendix 6.4, which states the protocols in Section 6.1 are sound against an on-path attacker *on*$\mathcal{A}$ as long as NTP packets are authenticated, randomness is produced from a cryptographically-strong RNG, and honest parties initialize their $\mathtt{xmt}_j$ variables for each server to a 64-bit number generated by their RNG upon reboot. For the protocol described in Figure 6, 7, which randomizes the 32-bit sub-second granularity of the expected origin timestamp, we also require that the second-level granularity of the client's local time t_c isn't replicated too often within NTP queries.

We suppose NTP is authenticated with MAC of length $2n$. Let $on\mathcal{A}$ be any on-path attacker, and let $\mathtt{ts}$ be any transcript involving a maximum of s trusted servers per client and a maximum of τ exchanges involving any single client-server pair that replicate any t_c value (up to the second) at most γ times. Let i^* be a client who does not query $on\mathcal{A}$ as server.

As before, we use a sequence of games to prove that $on\mathcal{A}$ tampers the state of $\mathcal{P}_{i^*}$ with probability at most $\epsilon_{on\mathcal{A}}$. We start by reusing games G_0 and G_1 from the proof of Theorem 61 above. It is straightforward to validate that the reduction between those games continues to hold against an on-path attacker. We now build a new sequence of games that (1) reflects $on\mathcal{A}$'s ability to view the contents of honest parties' messages and (2) utilizes the MAC tag to limit $on\mathcal{A}$'s spoofing capacity.

Foreshadowing the end of the proof, we will use Lemma F1 to arrive at the final bound. As such, our analysis simply describes the impact of each game on the probability of success during an NTP exchange (Q_d) and between NTP exchanges (Q_b).

Game G'_2. This game is identical to G_1, except that the network $\mathcal{N}$ is additionally instructed to drop all the packets sent by $on\mathcal{A}$ that are simply 'preplays' of packets sent between honest parties; *i.e.,* spoofed packets sent by $on\mathcal{A}$ that are identical to existing packets in $\mathcal{N}$'s queue such that $on\mathcal{A}$'s packet will be delivered first.

Recall that our definition of soundness is agnostic to preplay attacks. Hence, forbidding them has no effect on the adversary's success probability, *i.e.,* $\Pr^1_{on\mathcal{A}} = \Pr^{2'}_{on\mathcal{A}}$.

Game G'_3. This game is identical to G'_2, except that we abort the execution if the client $\mathcal{P}_{i^*}$ sends two different queries to the same server with identical origin timestamps. We stress that this constraint is independent of $on\mathcal{A}$'s behavior.

Consider a single NTP exchange between client $\mathcal{P}_{i^*}$ and server $\mathcal{P}_j$ where the client's clock begins at t_c. $\mathcal{P}_{i^*}$'s origin timestamp replicates a previous choice with probability at most $q_{32} = 2^{-32}\gamma$ if only the sub-second granularity is randomized or $q_{64} = 2^{-64}\tau$ if the entire expected origin timestamp is randomized. If the honest client repeats an origin timestamp, then $on\mathcal{A}$ may trivially attack an NTP exchange by replaying (an already-MAC'd) responses from previous exchanges.

Hence, the transformation from game G'_2 to game G'_3 affects Q_d by at most q_E. We remark that $on\mathcal{A}$ only requires 1 packet to perform this attack, so the resulting probability is independent of the bandwidth R.

Game G'_4. This game is identical to G'_3, except that the network $\mathcal{N}$ is instructed to drop all of $on\mathcal{A}$'s 'replayed' packets, *i.e.,* packets sent by $on\mathcal{A}$ that are identical to prior packets sent between honest parties and (1) have already been delivered or (2) are in $\mathcal{N}$'s queue for delivery before *on*$\mathcal{A}$'s packet. These replayed packets will have valid MAC tags but stale origin timestamps.

Consider what happens when a response packet from server $\mathcal{P}_j$ is replayed to target $\mathcal{P}_{i^*}$, or when a query packet from target $\mathcal{P}_{i^*}$ is replayed to server $\mathcal{P}_j$ and elicits $\mathcal{P}_j$'s legitimate response packet. If $\mathcal{P}_{i^*}$ and $\mathcal{P}_j$ are currently engaged in an NTP exchange, then $\mathcal{P}_{i^*}$'s state variable $\texttt{xmt}_j$ is set to an origin timestamp that is distinct from the one in the replay packet, so the replayed packet definitely fails `TEST2` by the constraint imposed by Game G'_3. On the other hand, if $\mathcal{P}_{i^*}$ and $\mathcal{P}_j$ are between exchanges, then $\texttt{xmt}_j$ is set to a randomized value with 64 bits of entropy. Hence, the transformation from game G'_3 to game G'_4 affects Q_b by at most 2^{-64}.

Game G'_5. This game is identical to G'_4, except that the network $\mathcal{N}$ consciously corrupts all MAC tags in *off*$\mathcal{A}$'s spoofed packets that aren't replays or preplays, so they never verify. We note that *on*$\mathcal{A}$ has no chance of winning game G'_5 (that is, $\Pr^{5'}_{on\mathcal{A}} = 0$, and thus $Q_d = Q_b = 0$ for game G'_5) because all of the packets she sends are rejected by their recipients for having invalid tags. Hence, *on*$\mathcal{A}$ cannot get any honest party to read its spoofed packets, much less change their state as a result of them. Additionally, we claim that the Game $G'_4 \to G'_5$ transformation affects Q_d by $\frac{R\varrho}{360+n} \cdot \mathrm{Adv(EU\text{-}CMA)}$ and Q_b by $2^{-64} \cdot \mathrm{Adv(EU\text{-}CMA)}$.

To prove the claim, we replace the tags of all *on*$\mathcal{A}$'s packets toward server $\mathcal{P}_j$ or client $\mathcal{P}_{i^*}$ that aren't replays or preplays with an invalid tag $\perp$. We do this one packet at a time, starting with the final packet and working our way back up to the first one. By a simple hybrid argument, we see that each change has an impact with probability at most Adv(EU-CMA).

During an exchange, a single forged MAC permits the attacker to respond to a query with timing data of her own choosing, and a simple union bound gives the bound on Q_d stated above. In between exchanges, a forged message must also include the origin timestamp matching $\mathcal{P}_{i^*}$'s randomly-chosen $\texttt{xmt}_j$ or else the packet will fail `TEST2`, yielding the bound on Q_b.

Putting it all together. Game G_1 additively impacts $\epsilon_{on\mathcal{A}}$, and Game G'_2 has no effect. Games G'_3, G'_4, and G'_5 all depend on k, and they detail the combined vulnerability of NTP to an on-path attacker during and between exchanges:

$$\begin{aligned} Q_d &\leq q_E + \frac{R\varrho}{360+n} \cdot \mathrm{Adv(EU\text{-}CMA)} \\ Q_b &\leq 2^{-64} \cdot [1 + \mathrm{Adv(EU\text{-}CMA)}] \approx 2^{-64}, \end{aligned}$$

where the final approximation follows from the fact that $1 + \mathrm{Adv(EU\text{-}CMA)} \approx 1$ for any reasonable MAC. Lemma F1 then bounds the probability that *on*$\mathcal{A}$ affects a particular client-server state k times in a row beginning from a specified starting point, and (as before) multiplying this value by the $s\tau$ possible starting points yields the bound in Theorem 62.