Improving the Performance and Security of the TOTD DNS64 Implementation

Gábor Lencse
Department of Telecommunications, Széchenyi István University
Egyetem tér 1, Győr, H-9026, HUNGARY, lencse@sze.hu
and
Sándor Répás
Department of Telecommunications, Széchenyi István University
Egyetem tér 1, Győr, H-9026, HUNGARY, repas.sandor@sze.hu

ABSTRACT

DNS64 and NAT64 IPv6 transition mechanisms are expected to play an important role in the near future to solve the problem that some of the new clients will not be able to get public IPv4 addresses and thus having only IPv6 addresses they still should be able to reach servers that have only IPv4 addresses. In our earlier experiments, the TOTD DNS64 implementation showed significantly better average performance than BIND, however TOTD was not stable, therefore now it was carefully tested to find the reason for its experienced strange behavior. Besides the detailed description of the testing method, the bug and the correction, a security vulnerability is disclosed and a patch is provided. The performance and the stability of the modified versions of TOTD are analyzed and compared to that of the original TOTD and BIND.

Keywords: IPv6 deployment, IPv6 transition solutions, performance analysis, DNS64, TOTD, security, cache poisoning attack, random permutation

1. INTRODUCTION

The depletion of the global IPv4 Address Pool[1] will be a driving force for the deployment of IPv6 in the forthcoming years. The internet service providers (ISPs) can still supply the relatively small number of new Internet servers with IPv4 addresses from their own pool but the huge number of new clients can get IPv6 addresses only. However, the vast majority of the Internet sites still uses IPv4 only. Thus from the many issues of the co-existence of IPv4 and IPv6, the communication of an IPv6 only client with an IPv4 only server is the first practical task to solve in the upcoming phase of the IPv6 deployment. The authors believe that DNS64 [1] and NAT64 [2] are the best available techniques that make it possible for an IPv6 only client to communicate with an IPv4 only server. (The operation of DNS64 and NAT64 will be introduced in section 2.) There are a number of implementations for both DNS64 and NAT64. When a network operator decides to support DNS64 and NAT64, it can be a difficult task to choose the right implementations because there can be security, reliability and performance issues. Several papers were published in the topic of performance analysis of different DNS64 and NAT64 implementations. We have given a short survey of them in our previous paper about the performance analysis and comparison of different DNS64 implementations for Linux, OpenBSD and FreeBSD [3]. There we have pointed out that the average performance of TOTD was better than that of BIND but TOTD was not stable. We also concluded that TOTD deserves a thorough code review and a bugfix. We give an account of this work and its results in this paper.

[1]IANA delegated the last five “/8” IPv4 address blocks to the five Regional Internet Registries in 2011 [4], of which APNIC has already depleted its IPv4 address pool in 2011 and RIPE NCC did so in 2012 [5]. It means that APNIC and RIPE NCC must use a more strict allocation policy for their very last /8 block.

The remainder of this paper is organized as follows: first, the operation of the DNS64+NAT64 solution is described, second, TOTD is introduced, third, our way of debugging is presented including the description of the test network and the testing method and also the way the bug was found and eliminated, fourth, our modification for security enhancement is detailed, fifth, the performance and stability of the following four DNS64 implementations are evaluated: original TOTD, TOTD with bugfix, TOTD with bugfix plus security enhancement and BIND as a forwarder, and finally, our conclusions are given.

2. THE OPERATION OF DNS64 AND NAT64

To enable an IPv6 only client to connect to an IPv4 only server, one can use a DNS64 server and a NAT64 gateway. The DNS64 server should be set as the DNS server of the IPv6 only client. When the IPv6 only client tries to connect to any server, it sends a recursive query to the DNS64 server to find the IPv6 address of the given server. The DNS64 server uses the normal DNS system to find out the IPv6 address of the server.

- If the answer of the DNS system contains an IPv6 address then the DNS64 server simply returns the IPv6 address as its answer to the recursive query.
- If the answer of the DNS system contains no IPv6 address then the DNS64 server finds the IPv4 address (using the normal DNS system) and it constructs and returns a special IPv6 address called IPv4-Embedded IPv6 Address [6] containing the IPv4 address of the given server in the last 32 bits. In the first 96 bits, it may contain the NAT64 Well-known Prefix or a network specific prefix from the network of the client.

The route towards the network with the given IPv6 prefix should be set in the IPv6 only client (and in all of the routers along the route from the client to the NAT64 gateway) so that the packets go through the NAT64 gateway.

The IPv6 only client uses the received IPv6 address to communicate with the desired (IPv4 only) server. The traffic between the IPv6 only client and the IPv4 only server travels through the NAT64 gateway in both directions. The NAT64 gateway makes their communication possible by constructing and forwarding the appropriate version IP packets. For constructing IPv4 packets from IPv6 packets, the NAT64 gateway takes the IPv4 address of the server from the last 32 bits of the destination IPv6 address, which is actually an IPv4 embedded IPv6 address. In the opposite direction, the NAT64 gateway is able to determine the IPv4 address of the client by using stateful NAT.

For a more detailed but still easy to follow introduction, see [7] and for the most accurate and detailed information, see the relating RFCs: [1] and [2].

3. TOTD IN A NUTSHELL

TOTD is a lightweight DNS64 implementation that was written by Feike W. Dillema as a part of the 6net project [8]. TOTD is available in source code from GitHub [9].

Section 5.3.7 of [8] writes that:

"The 'Trick or Treat' DNS-ALG called 'totd' is a lightweight DNS proxy (not a full DNS server); it does not recursively resolve queries itself. It is lightweight in that it consists of less than 5000 lines of C code. […] The 'totd' DNS-ALG is a proxy that forwards queries to one or more real nameservers that will recursively resolve queries. Such nameservers are called forwarders in totd terminology. If there are multiple forwarders specified, it will try them in the order listed. As a proxy, totd sits between client resolvers and real forwarder nameservers and as such receives requests from clients which totd normally forwards to a real nameserver to resolve. When it subsequently receives a response from the nameserver it simply forwards it back to the client. […] If the nameserver that totd forwards the AAAA query to, does not return an IPv6 address for the AAAA query, totd will make a second query but this time for an A record of the hostname of the original query. The resulting IPv4 address is then used to construct a fake IPv6 address, by replacing the lower 32 bits of the specified prefix with this IPv4 address. The resulting IPv6 address is sent as response to the original AAAA record query."

As for its licence, different parts of the code has slightly different licences but all licenses are of BSD style except for the optional built in tiny web server called SWILL, which has a GPL 2.1 licence.

Due to the free software [10] also called open source [11] nature of TOTD, it was possible for us to look into the source code and also to add some more lines that enrich the debug info sent to syslog.

4. DEBUGGING TOTD

During our measurement for our previous paper [3], TOTD 1.5 was tested in high load situations. TOTD occasionally stopped responding for about a minute and continued the operation afterwards. It produced similar behavior under all the three operating systems (Linux, OpenBSD and FreeBSD).

Before starting our work, we contacted the author of TOTD, who was very helpful and gave us advices but he did not have enough free time to set up a testbed and debug the software himself. Thus we did so. We used nearly the same test network as in paper [3] with the addition of the parts necessary for monitoring the traffic of the DNS64 server (and the CPU and memory parameters of the client computers were somewhat different) and the description is taken from there, too.

The Structure and Operation of the Test Network

The test network was set up in the *Infocommunications Laboratory* of the Department of Telecommunications, Széchenyi István University. The topology of the network is shown in Fig. 1. The central element of the test network is the DNS64 server. For the measurements, we needed a namespace that:

- can be described systematically
- can be resolved to IPv4 only
- can be resolved without delay

The 10-{0..10}-{0..255}-{0..255}.zonat.tilb.sze.hu name space was used for this purpose. This namespace was mapped to the 10.0.0.0 – 10.10.255.255 IPv4 addresses by the `teacherb` authoritative name server.

The tested TOTD DNS64 server program running on the Intel PIII test computer mapped these IPv4 addresses to the IPv6 address range: 2001:738:2c01:8001:ffff::0a00:0000 – 2001:738:2c01:8001:ffff::0a0a:ffff.

The DELL IPv6 only workstations at the bottom of the figure played the role of the clients for the DNS64 measurements.

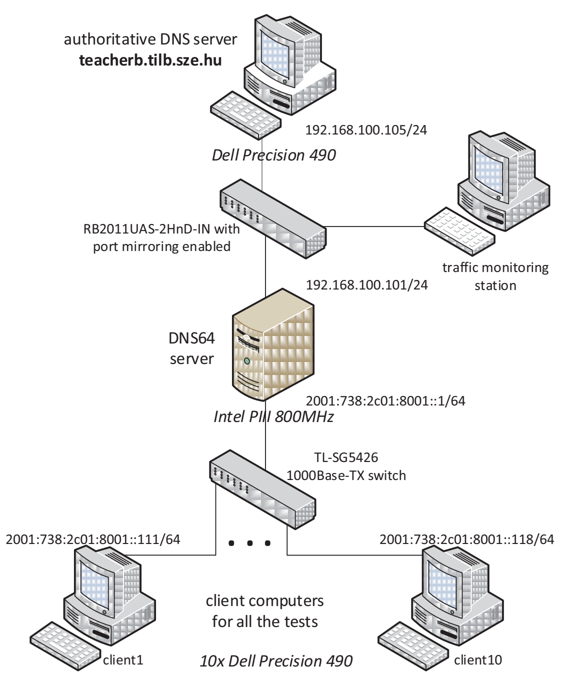

Figure 1. Topology of the DNS64 test network.

The switch between the PIII test computer executing TOTD and the `teacherb` authoritative name server was added to mirror the traffic to the monitoring station.

The Configuration of the Computers

A test computer with special configuration was put together for the purpose of the DNS64 server in order that the clients will be able to produce high enough load for overloading it. The CPU and memory parameters were chosen to be as little as possible from our available hardware base in order to be able to create an overload situation with a finite number of clients, and only the network cards were chosen to be fast enough. The configuration of the test computer was: Intel D815EE2U motherboard, 800MHz Intel Pentium III (Coppermine) processor, 128MB, 100MHz SDRAM, Two 3Com 3c940 Gigabit Ethernet NICs. For the 10 client computers and for the IPv4 DNS server, standard *DELL Precision Workstation 490* computers were used.

Debian Squeeze 6.0.3 GNU/Linux operating system was installed on all the computers with the exception of the authoritative DNS server (`teacherb.tilb.sze.hu`), which had Debian Wheezy 7.1 GNU/Linux. All the computers had 64 bit operating systems with the exception of the PIII test computer that had a 32 bit one.

IPv4 DNS Server Settings

The DNS server was a standard DELL Linux workstation using the 192.168.100.105 IP address and the symbolic name `teacherb.tilb.sze.hu`. BIND was used for authoritative name server purposes in all the DNS64 experiments. The version of BIND was 9.8.4-rpz2+rl005.12-P1 as this one can be found in the Debian Wheezy distribution.

The 10.0.0.0/16-10.10.0.0/16 IP address range was registered into the `zonat.tilb.sze.hu` zone with the appropriate symbolic names.

DNS64 Server Settings

The network interfaces of the freshly installed Debian Squeeze Linux operating system on the Pentium III computer were set according to Fig. 1.

In order to facilitate the IPv6 SLAAC (*Stateless Address Autoconfiguration*) of the clients, `radvd` (*Router Advertisement Daemon*) was installed on the test computer.

As TOTD is just a DNS forwarder and not a DNS recursor it was set to forward the queries to the BIND running on the `teacherb` computer.

The content of the `/etc/totd.conf` file was set as follows:

```
forwarder 192.168.100.105 port 53
prefix 2001:738:2c01:8001:ffff::
retry 300
```

Client Settings

Debian Squeeze was installed on the DELL computers used for client purposes, too. On these computers, the DNS64 server was set as the name server in the following way:

```
echo "nameserver 2001:738:2c01:8001::1" > \
      /etc/resolv.conf
```

DNS64 Performance Measurements

The CPU and memory consumption of the DNS64 server was measured in the function of the number of requests served. The measure of the load was set by starting test scripts on different number of client computers (1-10). In order to avoid the overlapping of the namespaces of the client requests (to eliminate the effect of the DNS caching), the requests from the number `i` client used target addresses from the 10.$i.0.0/16 network. In this way, every client could request 2^{16} different address resolutions. For the appropriate measurement of the execution time, 256 experiments were done and in every single experiment, 256 address resolutions were performed using the standard `host` Linux command. The execution time of the experiments was measured by the GNU `time` command. (Note that this command is different from the `time` command of the bash shell.)

The clients used the following script to execute the 256 experiments:

```
#!/bin/bash
i=`cat /etc/hostname|grep -o .$`
rm dns64-$i.txt
do
    for b in {0..255}
    do
        /usr/bin/time -f "%E" -o dns64-$i.txt \
           -a ./dns-st-c.sh $i $b
    done
done
```

The *synchronized start* of the client scripts was done by using the “Send Input to All Sessions” function of the terminal program of KDE (called `Konsole`).

The `dns-st-c.sh` script (taking two parameters) was responsible for executing a single experiment with the resolution of 256 symbolic names:

```
#!/bin/bash
for c in {0..252..4} # that is 64 iterations
do
    host –t AAAA 10-$1-$2-$c.zonat.tilb.sze.hu &
    host –t AAAA \
        10-$1-$2-$((c+1)).zonat.tilb.sze.hu &
    host –t AAAA \
        10-$1-$2-$((c+2)).zonat.tilb.sze.hu &
    host –t AAAA \
        10-$1-$2-$((c+3)).zonat.tilb.sze.hu
done
```

In every iteration of the `for` cycle, four `host` commands were started, from which the first three were started asynchronously (“in the background”) that is, the four commands were running in (quasi) parallel; and the core of the cycle was executed 64 times, so altogether 256 host commands were executed. (The client computers had two dual core CPUs that is why four commands were executed in parallel to generate higher load.)

In the series of measurements, the number of clients was increased from one to ten and the time of the DNS resolution was measured. The *CPU and memory utilization* were also measured on the test computer running DNS64. The following command line was used:

```
nice -n -10 dstat -t -T -c -m -n -N eth1,eth2 \
  -i -I 21,22 -p --unix --output load.csv
```

Experiments and Observations

First, we expected to face with some resource exhaustion problem thus we performed several runs of the experiments. We checked that the `teacherb` authoritative name server was always responsive and *sometimes* TOTD stopped responding. However we could not find any regularity in its behavior.

When we specified two forwarders, we found that TOTD stopped responding only for a few seconds (instead of a minute or so) and it did so if the same forwarder was specified twice.

Thinking of resource exhaustion, various limits were raised as follows:

```
echo "512000">/proc/sys/net/core/rmem_max
echo "512000">/proc/sys/net/core/wmem_max
echo "512000">/proc/sys/net/core/rmem_default
echo "512000">/proc/sys/net/core/wmem_default
echo "512000 800000 1000000" \
    > /proc/sys/net/ipv4/udp_mem
echo "65536">/proc/sys/net/ipv4/udp_rmem_min
echo "65536">/proc/sys/net/ipv4/udp_wmem_min
echo "128000" \
    > /proc/sys/net/core/netdev_max_backlog
```

We also checked the number of open sockets (still thinking of resource exhaustion) but this was also not the problem. Using the command line below, we monitored everything that seemed to be reasonable:

```
dstat -t -c -m -l -i -p -d -g -r -s -y --aio \
  --fs --ipc --lock --socket --tcp --udp \
  --unix --vm -n --unix --output test.csv
```

We still did not find the reason of the strange “stopping” phenomenon, but having more and more experiences with TOTD, we felt that sometimes the DNS messages “just disappear” probably between the PIII test computer and the `teacherb` authoritative DNS server. Therefore, we decided to trace them very accurately. The switch between the PIII test computer and the authoritative DNS server was set to mirror the traffic to the monitoring station, where it was captured and recorded by `tshark`. We also added some more detailed logging of the important events (e.g. the arrival of the clients’ requests, the sending of the query to the authoritative name server and the arrival of its answer, etc.) to the source code of TOTD with nanosecond accuracy timestamps using the `clock_gettime()` C library function and the clocks of all the computers (the clients, the PIII test computer, `teacherb` and the monitoring station) were synchronized by NTP.

In TOTD, we also decoded and logged the content of the DNS messages. This led us finally to the bug. Both DNS queries and responses begin by a 16 bit *Transaction ID* field set by the client and returned by the server. It lets the client match responses to requests. (See page 521 of [12].) As TOTD acts as a proxy, after receiving a query from a client it must behave as a client itself: it has to send a query to the DNS server that resolves recursive queries (called forwarder in the TOTD terminology). TOTD must generate a transaction ID for the query. We have checked the C code for transaction ID generation in file `ne_mesg.c`. It was done by a function named **`mesg_id()`** containing a static variable "**`id`**":

```
uint16_t mesg_id (void) {
    static uint16_t id = 0;

    if (!id) {
        srandom (time (NULL));
        id = random ();
    }
    id++;

    if (T.debug > 4)
        syslog (LOG_DEBUG,"mesg_id() = %d",id);
    return id;
}
```

The intention of the programmer seems to be clear: the value of the static variable is chosen randomly at the time of the first execution of the function and it is incremented by one at any later execution.

However the actual behavior of the C code above is slightly different. When the value of **`id`** is 0xffff and it is incremented then its value will be 0 and it is returned normally but at the next execution of the function the value of **`id`** will not be incremented to 1 rather it will be randomized again. Its value may be close to 0xffff with a small but positive probability. It means that the transaction ID of a new query may be equal to that of another recently sent query that is still waiting for the response. And it will cause problem when matching the responses to the queries. We have provided a quick bugfix by checking if the value of **`id`** became 0 and if it was so then promptly increasing it to 1 in order to avoid the hazardous re-randomization:

```
uint16_t mesg_id (void) {
    static uint16_t id = 0;

    if (!id) {
        srandom (time (NULL));
        id = random ();
    }
    if ( !++id ) ++id; /*correction for id==0*/

    if (T.debug > 4)
        syslog (LOG_DEBUG,"mesg_id() = %d",id);
    return id;
}
```

After this modification, TOTD remained responsive even during a whole night testing.

5. OUR SECURITY ENHANCEMENT FOR TOTD

The Problem of Sequential Transaction IDs

TOTD uses sequential transaction IDs, which are trivial to predict thus TOTD is vulnerable to cache poisoning using the transaction ID prediction attack. (The attacker persuades the victim DNS server to send a query for a given domain name and sends a response before the response of the authoritative name server by predicting the transaction ID and the victim DNS server will accept its answer, see section 4 of [13] for more details.) This is a serious security threat for all the DNS servers. The most widely used BIND [14] used also sequential transaction IDs prior to version 8.2, pseudorandom transaction IDs were introduced in v8.2 and further enhancements were made in v9, but it still has certain vulnerabilities according to [15].

Our Solution

As for TOTD, we also chose the randomization of the transaction IDs. However, the naive use of individually generated random IDs would require keeping a record of them if they are still in use, which would make necessary the modification of the source code of TOTD at multiple places (e.g. to register them when a query is made and to delete them when an answer is received or the time-out value expired). Therefore, the method of pre-generated random permutations was chosen to be able to keep the modifications within a single file. Two further considerations were made:

- Using up all the 65536 elements of a random permutation would result in predictability when we are approaching to the exhaustion of the pool.
- The lastly used elements of a given permutation may still be in use when some of them may appear between the first elements of a new permutation (resulting in the same bug that was fixed recently).

Therefore, our solution uses two alternating pools for the transaction IDs, 0-0x7fff and 0x8000-0xffff and only the first half of the elements of the permutations are used as transaction IDs. This design gives a reasonable security enhancement over the original situation for the price of a small amount of programming work (actually two functions in a single source file) and also a little waste of CPU power (50% of the elements of every permutation are not used).

Our Implementation

For preparing random permutation, the so-called "inside out" version of [16] was used. While the original version makes a random shuffle of the elements of an array in place and thus it requires the pre-initialization of the array, the modified version does both in a single cycle. The complexities of both the original and the modified algorithm are $O(N)$ where N denotes the size of the array, but the "inside out" version spares the work of the initialization and the exchange of the elements. Our commented C code can be seen in Fig. 2. Comments and recommendations for further improvements are especially welcome!

6. PERFORMANCE MEASUREMENTS AND COMPARISON

Even though the performance measurement results of the original TOTD and BIND can be found in [3] the measurements were repeated for the following reasons:

- Our previous paper is not open access therefore some readers of this paper may not have access to the results provided in that.
- The configuration of the client computers were slightly changed, which made the new results incomparable (or at least not fully comparable) with the old ones.
- The old measurement script did not use the **`-t AAAA`** option of the **`host`** command, thus then some other identifiers (e.g. MX records) were also requested but now we wanted to focus solely on the performance providing the AAAA records.

The measurements were taken the same way as they were done for bug hunting but the monitoring elements of the test network were removed. See the appropriate subsections of section 4 for the details. As now BIND was also included as DNS64 server, now we give its settings before the results of the different tested implementations.

The Set up of the BIND DNS64 Server

BIND version 9.9.1-P1 was compiled from source. The 2001:738:2c01:8001:ffff::/96 (network specific) prefix was set to BIND for the DNS64 function using the **`dns64`** option in the file **`/etc/bind/named.conf.options`**.

```
#define ARRAY_SIZE 0x8000        /* Size of the static array storing permutations */
#define NUM_USE 0x4000   /* Number of elements used up from the array of permutations */
#define LOW_START 0x0000 /* Starting value of the lower range */
#define HIGH_START 0x8000        /* Starting value of the higher range */

static uint16_t permutation[ARRAY_SIZE]; /* The static array for storing random permutations */

/* Prepare a random pemutation of the integers [start, start+ARRAY_SIZE-1] into the static array */
/* Algorithm: http://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle#The_.22inside-out.22_algorithm*/
void make_random_permutation(int start) {
        int i,j;

        permutation[0]=start;
        for (i=1; i<ARRAY_SIZE; i++) {
                j=random()*(double)(i+1)/RAND_MAX; /* random number from the range [0, i] */
                if ( j != i )
                        permutation[i]=permutation[j];
                permutation[j]=start+i;
        }
}

/* Provide hard to predict unique random DNS Transaction IDs */
/* by using random permutations and alternating ranges */
uint16_t mesg_id (void) {
        static int range=0; /* indicates that no permutation is generated yet */
        static int index;

        if ( !range ) {
                srandom(time(0));        /* initialize random number generator seed */
                range=1;         /* choose the lower range first */
                make_random_permutation(LOW_START);
                index=0;         /* start from the first element */
        }
        if ( index == NUM_USE ) {/* if the pool is exhasuted */
                if ( range == 1 ) {
                        range=2; /* choose the higher range */
                        make_random_permutation(HIGH_START);
                }
                else {
                        range=1; /* choose the lower range */
                        make_random_permutation(LOW_START);
                }
                index=0;
        }
        if (T.debug > 4)
                syslog (LOG_DEBUG, "mesg_id() = %d", permutation[index]);
        return permutation[index++];
}
```

Figure 2. The modifications made to the **ne_mesg.c** source file.

BIND is able to operate as a recursor, but to make the performance of BIND and TOTD comparable, BIND was set as a forwarder. It was done by the following additional settings in the **named.conf** file:

```
forwarders { 192.168.100.105; };
forward only;
```

The Interpretation of the Tables with the Results

Even though the graphical representation may ease the comparison of the results in many cases, now we preferred the table format because of the following reasons:

- The maximum value of the response time of the unmodified TOTD was more than one order of magnitude higher than that of the others thus a comparison chart would be ill looking.
- There were many types of characteristics measured thus tables were found more efficient than using different diagrams for every types of values.

For the synoptic view and easy comparison of the results, the four tables were put on the same page.

Each table is to be interpreted as follows. The first row shows the number of the clients. (The offered load was proportional with the number of the clients.) Rows 2, 3 and 4 show the maximum value, the average and the standard deviation of the execution time of one experiment, respectively. An experiment is an execution of the **dns-st-c.sh** script, which one executes the "**host -t AAAA**" command 256 times using different DNS names to eliminate the effect of caching. And this script was also executed 256 times by each client that took part in the measurement. Rows 5 and 6 show the average and the standard deviation of the CPU utilization of the PIII test computer running the DNS64 service. Row 7 shows the memory consumption of the DNS64 service. Note that this value could be measured with high uncertainty because it was measured by the change of the free memory on the test computer. Row 8 shows the number of requests processed per seconds.

TABLE I. PERFORMANCE OF THE UNMODIFIED TOTD 1.5

Number of clients		1	2	3	4	5	6	7	8	9	10
Exec. time of 256 **`host -t AAAA`** commands (s)	max	0.54	0.64	0.84	117.80	17.07	83.37	25.63	62.74	50.08	31.48
	average	0.524	0.588	0.770	1.879	1.296	1.862	1.857	2.290	2.474	2.673
	std. dev.	0.007	0.021	0.041	9.096	0.932	4.689	1.419	3.594	2.818	1.634
CPU utilization (%)	average	22.33	57.89	92.02	50.94	95.29	83.49	94.87	87.97	92.18	96.06
	std. dev.	4.24	3.25	2.43	48.27	20.33	36.69	21.50	32.00	26.38	19.06
Memory consumption (kB)		892	852	964	1624	1372	1624	1156	1724	1780	1992
Performance (request/s)		488	870	997	545	988	825	965	894	931	958

TABLE II. PERFORMANCE OF TOTD 1.5 WITH SEQUENTIAL TRANSACTION IDS

Number of clients		1	2	3	4	5	6	7	8	9	10
Exec. time of 256 **`host -t AAAA`** commands (s)	max	0.530	0.630	0.830	1.060	1.330	1.600	1.830	7.000	7.180	7.460
	average	0.518	0.595	0.759	1.002	1.239	1.498	1.762	2.055	2.268	2.535
	std. dev.	0.007	0.015	0.039	0.042	0.060	0.075	0.068	0.160	0.152	0.142
CPU utilization (%)	average	18.94	59.29	90.99	99.22	99.91	99.99	99.99	99.99	100.00	100.00
	std. dev.	4.14	3.96	2.32	0.76	0.30	0.13	0.10	0.13	0.04	0.00
Memory consumption (kB)		808	944	1076	1188	1232	1484	1556	1752	1780	1768
Performance (request/s)		494	860	1011	1022	1033	1026	1017	997	1016	1010

TABLE III. PERFORMANCE OF THE TOTD 1.5 WITH PSEUDORANDOM TRANSACTION IDS

Number of clients		1	2	3	4	5	6	7	8	9	10
Exec. time of 256 **`host -t AAAA`** commands (s)	max	0.540	0.660	0.900	1.090	1.340	1.640	1.890	2.090	7.230	2.710
	average	0.520	0.594	0.760	1.017	1.253	1.536	1.812	2.029	2.279	2.574
	std. dev.	0.007	0.021	0.041	0.041	0.060	0.078	0.087	0.053	0.178	0.081
CPU utilization (%)	average	20.81	59.49	91.25	98.86	99.89	99.98	100.00	100.00	99.95	100.00
	std. dev.	3.81	2.47	2.20	1.00	0.32	0.14	0.05	0.00	0.72	0.00
Memory consumption (kB)		808	1004	1116	1300	1232	1540	1724	1788	1892	1992
Performance (request/s)		492	862	1009	1007	1022	1000	989	1009	1011	995

TABLE IV. PERFORMANCE OF BIND AS A FORWARDER

Number of clients		1	2	3	4	5	6	7	8	9	10
Exec. time of 256 **`host -t AAAA`** commands (s)	max	2.820	1.070	1.690	2.360	2.880	3.480	3.960	4.690	5.220	5.830
	average	0.831	1.041	1.588	2.246	2.785	3.390	3.839	4.550	5.086	5.719
	std. dev.	0.125	0.010	0.047	0.060	0.065	0.049	0.101	0.097	0.095	0.086
CPU utilization (%)	average	58.27	98.85	99.94	100.00	100.00	100.00	100.00	100.00	100.00	100.00
	std. dev.	5.30	0.81	0.24	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Memory consumption (kB)		24348	48528	53624	51312	53132	51336	52876	50888	51420	50384
Performance (request/s)		308	492	484	456	460	453	467	450	453	448

Performance Results of the Original TOTD

The results of the unmodified TOTD 1.5 are shown in table I. The maximum values of the execution time fluctuate in a huge range and show randomness. The average execution time grows with the load, except for the experiments with four clients, which was an "unlucky" case – see the maximum value of the execution time (117.8s), the low average CPU utilization (50.94%) and low performance (545 requests per seconds). This is why it was so important to find the bug in TOTD.

Performance Results of TOTD with a Bugfix

Table II shows the performance of TOTD with the quick bugfix that made the transaction IDs truly sequential by eliminating the hazardous re-randomization. Even though TOTD was put under very serious overload, it proved to be stable. The CPU was practically fully utilized at four clients (99.22%) and as the load was increased, TOTD did not collapse but the average response time increased only proportionally with the load. The memory consumption was also very low and showed only a very little growth in the function of the load. The number of requests processed per second approximated its maximum value at 3 clients and could not significantly increase because of the lack of free CPU capacity but it did not show significant decrease even in very serious overload situations thus TOTD complied with the graceful degradation principles [17].

Performance Results of TOTD with Enhanced Security

Table III shows the performance of TOTD with pseudorandom transaction IDs. The results are really good news: TOTD shows no visible decrease of the performance compared to the previous case: the graph displaying the average response time of this version of TOTD is hiding the graph of the previous version in Fig. 3. Therefore we recommend the application of this version of TOTD because the performance price of the increased security is marginal.

Performance Results of BIND as a Forwarder

Table IV shows the performance of BIND which was set up as a forwarder too (for fair comparison). We can lay down that BIND is also stable. Its memory consumption is significantly higher than that of TOTD. It is probably caused by caching. The bad news for BIND is its higher CPU time consumption. It uses much more computing power (58.27%) than TOTD with pseudorandom transaction IDs (20.81%) even under the load of one client. BIND reaches its maximum performance (492 requests per second) at two clients. Under serious overload (three or more clients), TOTD with pseudorandom transaction ID could process twice as many requests as BIND providing the half of the average response time of BIND. Thus TOTD with pseudorandom transaction IDs proved to be a good alternative of BIND.

Modification of the source code of TOTD
Our security enhancement patch for the `ne_mesg.c` file has been included into the source code of TOTD 1.5.3 [9].

7. CONCLUSIONS

A testbed was set up and the promising TOTD DNS64 server was tested extensively. The bug was found in its sequential transaction ID generation function and it was eliminated.

Because of its use of sequential transaction IDs, TOTD was found to be vulnerable to cache poisoning using the transaction ID prediction attack. This vulnerability was patched by a very computation efficient solution using random permutations and alternating ranges. The performance price of the increased security was found to be practically invisible. Therefore we recommend the application of this version of TOTD.

Under serious overload conditions, TOTD with our pseudorandom transaction ID generation could process twice as many requests than BIND providing 50% less average response time than BIND.

We conclude that TOTD with our pseudorandom transaction ID generation patch is a good candidate to be a number one DNS64 server solution for the transition period of the upcoming IPv6 deployment.

Our pseudorandom transaction ID generation patch has been included into the 1.5.3 version of the source code of TOTD on GitHub.

8. ACKNOWLEDGMENTS

The work of Gábor Lencse was supported by the TÁMOP-4.2.2.C-11/1/KONV-2012-0012: "Smarter Transport" – IT for co-operative transport system project. – The Project is supported by the Hungarian Government and co-financed by the European Social Fund.

The work of Sándor Répás was supported in the framework of TÁMOP 4.2.4. A/2-11-1-2012-0001 „National Excellence Program – Elaborating and operating an inland student and researcher personal support system convergence program" The project was subsidized by the European Union and co-financed by the European Social Fund.

The authors express their thanks to Gábor Horváth (Department of Networked Systems and Services, Budapest University of Technology and Economics) for suggesting the idea of using random permutations.

The authors appreciate the help and cooperation of Feike Willem Dillema, the author of TOTD.

Certain parts of the introduction and of the description of the performance testing were taken from our conference paper [3].

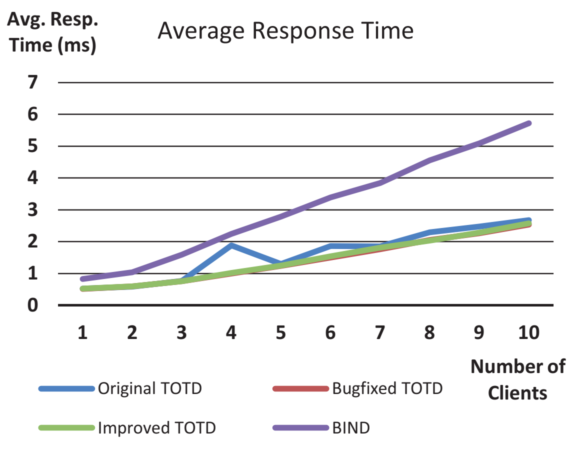

Figure 3. Comparison of the average response time of the different DNS64 implementations (lower is better)

9. REFERENCES

[1] M. Bagnulo, A Sullivan, P. Matthews and I. Beijnum, "DNS64: DNS extensions for network address translation from IPv6 clients to IPv4 servers", IETF, April 2011. ISSN: 2070-1721 (RFC 6147)

[2] M. Bagnulo, P. Matthews and I. Beijnum, "Stateful NAT64: Network address and protocol translation from IPv6 clients to IPv4 servers", IETF, April 2011. ISSN: 2070-1721 (RFC 6146)

[3] G. Lencse and S. Répás, "Performance Analysis and Comparison of Different DNS64 Implementations for Linux, OpenBSD and FreeBSD", Proceedings of the IEEE 27th International Conference on Advanced Information Networking and Applications (AINA 2013), (Barcelona, Spain, 2013. March, 25-28.) IEEE Computer Society, pp. 877-884. doi:10.1109/AINA.2013.80

[4] The Number Resource Organization, "Free pool of IPv4 address space depleted", http://www.nro.net/news/ipv4-free-pool-depleted

[5] RIPE NCC, "RIPE NCC begins to allocate IPv4 address space from the last /8", http://www.ripe.net/internet-coordination/news/ripe-ncc-begins-to-allocate-ipv4-address-space-from-the-last-8

[6] C. Bao, C. Huitema, M. Bagnulo, M Boucadair and X. Li, "IPv6 addressing of IPv4/IPv6 translators", IETF, October 2010. ISSN: 2070-1721 (RFC 6052)

[7] M. Bagnulo, A. Garcia-Martinez and I. Van Beijnum, "The NAT64/DNS64 tool suite for IPv6 transition", IEEE Communications Magazine, vol. 50, no. 7, July 2012, pp. 177-183.

[8] The 6NET Consortium, "An IPv6 Deployment Guide", Edited by Martin Dunmore, September, 2005 http://www.6net.org/book/deployment-guide.pdf

[9] TOTD source code at GitHub, https://github.com/fwdillema/totd.git

[10] Free Software Foundation, "The free software definition", http://www.gnu.org/philosophy/free-sw.en.html

[11] Open Source Initiative, "The open source definition", http://opensource.org/docs/osd

[12] Kevin R. Fall and W. Richard Stevens, TCP/IP Illustrated, Volume 1: The Protocols, Second Edition, Addison-Wesley Professional Computing Series, Second printing, May, 2012.

[13] A. Hubert and R. van Mook, "Measures for Making DNS More Resilient against Forged Answers", IETF, January 2009. (RFC 5452)

[14] Internet Systems Consortium, "Berkeley Internet Name Daemon (BIND)", https://www.isc.org/software/bind

[15] A. Klein, "BIND8 DNS Cache Poisoning – And a theoretic DNS cache poisoning attack against the latest BIND 9", Trusteer, July-August 2007, http://packetstorm.wowhacker.com/papers/attack/BIND_8_DNS_Cache_Poisoning.pdf

[16] R. Durstenfeld, "Algorithm 235: Random permutation", Communications of the ACM, Vol. 7 No. 7, (July 1964) p. 420. doi:10.1145/364520.364540

[17] NTIA ITS, "Definition of 'graceful degradation' ", http://www.its.bldrdoc.gov/fs-1037/dir-017/_2479.htm

Received: November 2013. Accepted: January 2014.