EAI Endorsed Transactions
on Security and Safety

Research Article EAI.EU

# An On-Demand Defense Scheme Against DNS Cache Poisoning Attacks

Zheng Wang[1,*], Shui Yu[2], and Scott Rose[1]

[1]National Institute of Standards and Technology, Gaithersburg, MD 20899, USA
[2]School of Information Technology, Deakin University, Burwood, VIC 3125, Australia

## Abstract

The threats of caching poisoning attacks largely stimulate the deployment of DNSSEC. Being a strong but demanding cryptographical defense, DNSSEC has its universal adoption predicted to go through a lengthy transition. Thus the DNSSEC practitioners call for a secure yet lightweight solution to speed up DNSSEC deployment while offering an acceptable DNSSEC-like defense. This paper proposes a new On-Demand Defense (ODD) scheme against cache poisoning attacks, still using but lightly using DNSSEC. In the solution, DNS operates in DNSSEC-oblivious mode unless a potential attack is detected and triggers a switch to DNSSEC-aware mode. The modeling checking results demonstrate that only a small DNSSEC query load is needed by the ODD scheme to ensure a small enough cache poisoning success rate.

Received on 23 April 2018; accepted on 09 March 2018; published on 15 May 2018

**Keywords:** DNS Security Extensions, DNS cache poisoning, model checking, query load, success rate.

doi:10.4108/eai.15-5-2018.154771

## 1. Introduction

Domain Name System (DNS) is today's largest name resolution system in use. Most of Internet applications rely on DNS to translate human friendly names to addresses, services, servers, etc. However, the early design of DNS did not pay sufficient attention to its security in 1980s. Thus the emerging security problems of DNS drove the community's efforts on developing DNS security mechanisms. One major progress on securing DNS is DNS Security Extensions (DNSSEC) [1], [2] as a set of specifications agreed by IETF in 2005. DNSSEC provides security capabilities by digitally signing DNS data using public-key cryptography. When a resolver issues a DNS query for a resource record in a DNSSEC-signed zone, the response includes not only the requested record but also the signatures for the record. The validity of the signed record can be determined via an authentication chain following the DNS hierarchy.

Unlike its standardization efforts, DNSSEC adoption had been debatable over years before 2008. Up until 2008, the community was split on the value of the DNSSEC effort ąłmany thought the deployment was quixotic, while a few others thought it was appropriate. However, the discovery of Kaminsky vulnerabilities [5] had been believed to change the debate on DNSSEC from "Do we really need DNSSEC?" to "How do we implement DNSSEC"? For a few years since 2008, the serious flaw in the DNS discovered by Dan Kaminsky prompted the Internet engineering community to grapple with what to do to remedy or fix it. While some temporary mitigating solutions were proposed in those years [3], [4], the community finally arrived at a consensus that the ultimate DNS security solution, namely DNSSEC, should roll out instead of tweaking the DNS protocols to address the Kaminsky bug as an interim step.

What security researcher Dan Kaminsky discovered in July 2008 is a DNS bug [5] that allows for cache poisoning attacks, where a hacker redirects traffic from a legitimate Web site to a fake one by injecting bogus DNS data without the user knowing. Thanks to digital signature provided in public key cryptography, DNSSEC strongly enough defends against hackers from

[*]Corresponding author. Email: zhengwang98@gmail.com

hijacking DNS transactions by allowing users to verify DNS data using digital signatures.

The strong defense against Kaminsky attacks provided by DNSSEC virtually takes effects only when DNSSEC is fully deployed across the Internet - from the DNS root zone at the top of the DNS hierarchy down to individual top-level domains (such as .com and .net), second-level domains, lower level domains, and even leaf domains in the DNS tree. Not only the target domain itself but also all its ancestor domains including the parent domain must be signed to ensure a complete trust chain to get protected by DNSSEC. Otherwise if only any domain in the trust chain turns DNSSEC oblivious, the target domain turns vulnerable to Kaminsky attacks. So the DNSSEC deployment is much like an "all or nothing" proposition. That is, incomplete or halfway DNSSEC deployment is likely to leave a much larger subset of the entire domain name space vulnerable than we may expect.

It is true that Internet scale DNSSEC deployment requires substantial costs, efforts, coordination, and time. Huston [6] measured the cost of DNSSEC deployment in terms of traffic load and resolution time. In his experiments, he observed that both cache resolver and authoritative name server suffer significant performance penalty when turning on DNSSEC. For a resolver as a DNSSEC validator, it takes 4 times longer in response time than and 8 times the traffic load compared to the non-DNSSEC case. For an authoritative name server serving a signed zone, the query response traffic for the signed zone have risen approximately 7-8 times the traffic for the unsigned zone. In another measurement on the DNSSEC overhead [7], the memory usage and the CPU usage are examined. The penalty for DNSSEC-enabled authoritative servers is a factor of two in terms of memory usage increase, and for cache resolvers, the increase is much more dramatic as over 4 times. For CPU usage increase, DNSSEC introduces an overhead of a factor of 1.1 to 2 to authoritative servers, leaving cryptographic operations such as zone signing for additional entities or equipment. And the cache resolver's CPU time for DNSSEC increases by a factor of 2.3 compared to DNS. The average packet size generated by DNSSEC is enlarged [8], [9]. Besides the performance penalty, as an Internet-scale cryptographic system, DNSSEC heavily relies on the secure key generation, storage, distribution and rollover as well as zone signing and record authentication operations. This make it necessary for a large amount of system reconstruction and rebuilding, which often need many hours of expensive human resources and hardware and software life cycle maintenance costs. Another important and indispensable aspect that may be underestimated is changing and increasing operational procedures and policies. DNSSEC poses new and higher demands for procedural controls that may materially affect: generation and protection of the private component of the key, secure export or import of the public components, and generation and signing of zone data. The personnel controls under DNSSEC are usually also enhanced to ensure personnel's proof of the requisite background, qualifications, and experience needed to perform their prospective DNSSEC specific and risk sensitive job responsibilities competently and satisfactorily.

Hence despite that the community has been sparing no efforts to use its resources to encourage DNS registries, ISPs and enterprises to upgrade to DNSSEC, global DNSSEC adoption is still well underway today. The positive side is that the root and 1407 top-level domains out of 1543 has rolled out DNSSEC [10]. But the rare adoption on lower level domains, which are mostly the ultimate destination of individual DNS lookups, might shed light on a low level of optimism on the universal DNSSEC adoption as a whole. Hence whether willing or not, there is no denying that DNSSEC hasn't been widely deployed yet, although it has been on the way for almost two decades. In some ways, the progress of DNSSEC deployment is similar to the undergoing IPv4 to IPv6 transition: slow and incremental.

Given that global DNSSEC deployment will not be seen in a short term largely due to concerns about its investment and costs, we propose in this paper a light-weighted countermeasure against cache poisoning attacks still using but lightly using DNSSEC. It changes DNSSEC operation from a model of persistent-defense to a model of detect-and-defense. In the solution, DNS operates in the DNSSEC-oblivious mode unless a potential attack is detected and triggers a switch to the DNSSEC-aware mode. Hence DNSSEC can be expected to be not so aggressively used. In this way, the unnecessary costs wasted by DNSSEC in the absence of attack are saved. The attack detection and the mode switch are basically dominated by recursive resolvers so that clients can be kept transparent to the countermeasure. Unlike today's DNSSEC practice which only allows ignorant and incompetent clients to make a decision on whether or not using DNSSEC, the proposed model of detect-and-defense makes full use of the detection capability of recursive resolvers to take up DNSSEC whenever needed. So clients can simply send DNSSEC-oblivious requests and count on their recursive resolvers to take appropriate DNSSEC defenses when necessary. Because of its efficiency and efficacy, the proposal can serve as an interim or transition mechanism for spreading and speeding DNSSEC adoption over a long-term transition. The rest of the paper is organized as follows: related work is presented in Section 2. The ODD scheme is elaborated in Section 3. The attack surface is discussed in Section 4. In Section 5, we present the performance analysis of the ODD scheme. Section 6 evaluates the ODD scheme through model checking. Finally, Section 7 concludes the paper.

## 2. Related Work

Before or in parallel with the DNSSEC rollout, there have been some proposals attempting to address the DNS cache poisoning risks in a light-weight way. As a non-DNSSEC solution to the DNS security, Fan et al. [11] proposed preventions embedded in security proxies. But their deployment costs are fairly high because security proxies need to be deployed at both authoritative servers and recursive resolvers to support packing and unpacking of all DNS packets with security label. A radical change to the existing DNS eco-system for tackling vulnerabilities in shared DNS resolvers [12]: removing shared DNS resolvers entirely and leaving recursive resolution to the clients. However, each individual client conducting its own resolutions may be still targeted by cache poisoning attackers. The solution also has a large performance penalty because the wide-area DNS traffic and DNS server load will grow significantly due to the absence of cache sharing. Sun et al. [13] proposed DepenDNS as a countermeasure which query multiple resolvers concurrently to verify a trustworthy answer. The reliability and availability of history response data used by DepenDNS is a great concern. Besides, the performance concern about DepenDNS is when the queries are multiplied, their processing overheads will also be multiplied. An extension to DNSSEC was proposed in [18], making the trust islands verifiable through extended chain of trust. Nevertheless, the overheads of DNSSEC are not lessened by the extension.

Shulman et al. [14] performed a critical study of the prominent defense mechanisms against poisoning attacks by off-path adversaries, concluding that existing easy-to-deploy defenses are not so reliable and thus transition to DNSSEC deserves the efforts. The capability of the DNS cache poisoning attacks was studied in [15] and [16], which are helpful to better understand our proposed defense.

## 3. ODD Scheme

Non-cryptography fixes or countermeasures against cache poisoning attacks are usually easy-to-deploy and light-weighted in terms of deployment costs. However, their vital weakness is insufficient defense compared with cryptography solutions such as DNSSEC. As analyzed above, the disadvantage of DNSSEC is its difficulty and costs in deployment. To "condense" DNSSEC as best as possible while retaining its security capability against cache poisoning attacks, we propose that DNSSEC can coalesce with other defense techniques such as attack detection. In the following, we will first discuss an attack detection scheme. Then the transition from the DNSSEC-oblivious mode to the DNSSEC-aware mode triggered by attack detection is presented.

### 3.1. Attack Detection

**Cache poisoning attacks.** Cache poisoning is where the attacker manages to inject bogus data into a resolver's cache with carefully crafted and timed DNS packets. A cache poisoned resolver will response with its wrongfully accepted and cached data, make its clients contact the wrong, and possibly malicious, servers. A resolver only accepts matching responses to its pending queries, and unexpected responses are simply ignored. A response packet is taken as "expected" and accepted by a resolver if and only if:

- The Question section of the reply packet matches the Question in the pending query;
- The response comes from the same network address to which the question was sent;
- The ID field of the reply packet matches that of the pending query;
- The response arrives on the same UDP port to which the question was sent;
- The authority and additional sections represent names that are within the same domain as the question: this is known as "bailiwick checking".

In order to have its bogus responses accepted by the target resolver, the attacker aims at matching all of the four sections in its bogus responses. Kaminsky proposed a class of effective attack schemes, among which the most mighty version is as follows:

1. The attacker sends a target DNS resolver a number of queries for a domain name. The domain name in question is random generated so as to be unlikely to be in the resolver's cache. The domain authoritative for the queried domain name is the target domain that the attacker tries to compromise.
2. The resolver thus issues requests to the real authoritative name servers for the domain name. In the mean time, attacker sends a flurry of forged responses to the target resolver. The responses delegates authority to the name servers owned by the attacker. In an attempt to be accepted by the target resolver, the forged response should guess

the matching sections in the genuine response such as source port number, source address, and transaction ID.

3. When one matching forged response is fed on the target resolver before the genuine response arrives, the target resolver accepts the forged one and discard the genuine one.

4. Any future requests for the domain names in the target domain will be directed to the bogus name server and thus responded with the bogus records. The attacker may append a long enough TTL (Time-To-Live) to the bogus delegation to reserve the bogus delegation in the cache as long as possible.

**Detection.** For the sake of being accepted by the target resolver, cache poisoning attackers have to guess the transaction ID, port number, and source address of the genuine response in their bogus responses. A number of bogus responses of guessing wrong are expected to be found by the target resolver before one bogus response may accidentally succeed. So failure response counting can be utilized to detect possible cache poisoning attacks.

For each specific domain name, failure responses are defined as those mismatches the combination of transaction ID, port number, and source address of the outstanding (wait-for-response) requests. In particular, failure responses are defined as any response attempt satisfying the following:

**a)** It matches the name in question (or more precisely, the triple <qname, qtype, qclass>) given in an outstanding query or a set of outstanding queries. Note that attackers usually exploits multiple outstanding queries for the same name to significantly increase the success rate of caching poisoning. This is referred to as "birthday attack". In that case, more than one outstanding queries may share one name in question.

**b)** If a) holds, it mismatches at least one item among transaction ID, port number, and source address of the name-matching outstanding query or any of the set of name-matching outstanding queries. Thus the resolver will not accept it as genuine.

As a means of attack detection, the resolver counts the incoming failure responses for any name in question in the outstanding queries until the count amounts to a Threshold of Defense (ToD).

The appropriate setting of ToD should take the following into considerations:

- A too large value will result in a non-negligible increase of cache poisoning success rate before defense measures come to effect. For example, Wang [15], [16] illustrated that the number of cache poisoning attempts is in the order of ten thousands to ensure a 50% chance of compromise in most cases of DNS operations. To secure the effect of defense, the value of the threshold should be no more than the order of ten thousands.

- A too small value will make it easier to trigger a defense. Problem of false positive stands here when non-malicious negligent users may unintentionally create a small amount of malformed responses which are identified as failure responses. When such false positive occurs, unnecessary defenses may be wasted due to the small threshold. Another exploit of a small threshold is that potential adversaries may intentionally feed a few failure responses on the target resolver in a bid to overload it with excessive defenses. Since the defense proposed as the following involves the DNSSEC-aware request resolved by the authoritative servers, excessive defenses will also cause substantial overhead of the authoritative servers. In that way, less efforts are needed for initiating such DoS like attacks on both the target resolver and the authoritative servers if the threshold is low.

## 3.2. Two Modes

**DNSSEC-oblivious mode.** In the DNSSEC-oblivious mode, the recursive resolver operates in compliant with the basic DNS. That is, it never sends DNSSEC requests or authenticate DNSSEC response unless it is explicitly required by the client (the client sets the DO bit in the request). In handling incoming responses to its outstanding queries, the recursive resolver simply checks and accepts them if they meet the five matches stated above. The only difference of the DNSSEC-oblivious mode from the basic DNS is that the DNSSEC-oblivious mode supplements the attack detection and thereby the transition to the DNSSEC-aware mode into the basic DNS.

The costs of the DNSSEC-oblivious mode are almost as low as the basic DNS. As long as no attack is detected, the DNSSEC-oblivious mode continues as a normalcy.

**DNSSEC-aware mode.** The DNSSEC-aware mode is transitioned from the DNSSEC-oblivious mode for some domain when that domain is hit by enough failure responses counted in the attack detection. The DNSSEC-aware mode is designed to use DNSSEC transactions to validate suspicious responses to the domain targeted by potential attackers.

The responding process in the DNSSEC-aware mode is illustrated in Fig. 1. When some domain is labeled as a suspicious target domain by the attack detection, the resolver should immediately initiate a separate

DNSSEC-aware request for that domain. That DNSSEC-aware request's response, which is called "validating response" hereinafter, is designated as the trustworthy authority for all upcoming responses to that domain. Thus all responses arriving prior to the arrival of the validating response for that domain, including those meet the aforementioned four matches, are simply hold on rather than accepted. Note that the hold-on responses may include a seemingly genuine but virtually bogus response. That bogus response looks like genuine because it meets the four matches. However, it is injected in a cache poisoning attempt which may accomplish such a success of response guessing after a number of failures. The hold-on responses may also contain the genuine response if it arrives earlier than the validating response.

```
$BogusCount \leftarrow 0$;
SEND THE REQUEST;
while $BogusCount < ToD$ do
    if time out then
        RETURN(TIME_OUT); % The DNS resolution times out
    LISTEN TO THE RESPONSE;
    if the response is bogus then
        $BogusCount \leftarrow BogusCount + 1$;
    else
        RETURN(THE RESPONSE); % The real response received
SEND THE DNSSEC REQUEST;
while not time out do
    LISTEN TO THE RESPONSE;
    if the response is validated then
        $ValidResponse \leftarrow$ the response;
    else if the response is genuine then
        $HoldonResponse \leftarrow HoldonResponse \cup \{$the response$\}$;
    if $(ValidResponse \neq \phi)$ & $(HoldonResponse \neq \phi)$ then
        if any response $\in HoldonResponse = ValidResponse$ then
            RETURN(THE RESPONSE); % The real response received
RETURN(TIME_OUT); % The DNS resolution times out
```

**Figure 1.** The responding process of DNSSEC–aware mode.

The validating response, if passing the DNSSEC validation, is deemed as trustworthy. It is compared against each response in the hold-on list. If there is some hold-on response matching the validating response, the transaction ends with returning the matching response to the client and discarding other hold-on responses, if any. Otherwise, if no candidate hold-on response survives the check for a match, all existing candidates will be discarded and the resolver will continue to wait for more candidate responses probably still to come until the resolution times out. Then each newly arrived responses, if any, will be checked against the validating response before it can be accepted and returned. In brief, the DNSSEC-aware mode attempts to return the first candidate response matching the validating response within the timeout period.

As can be seen from above, the DNSSEC-aware mode differs from the conventional DNS or DNSSEC specifications in the following:

- Unlike the conventional DNS specifications, the DNSSEC-aware mode does not attempt to make the resolver generate queries towards authoritative servers completely upon the client's requests (in case of cache miss). Instead, the resolver in the DNSSEC-aware mode produces specific separate DNSSEC queries as the security supplement to the conventional queries corresponding directly to the clients' requests. In other words, the DNSSEC-aware mode adds the "isolated" requests for validating response to the conventional DNS specifications.

- Unlike the conventional DNSSEC specifications, the DNSSEC-aware mode does not aggressively make DNSSEC queries for all domains in question. Instead, its DNSSEC transactions only cover those domains which are detected as suspicious victims of attacks.

- In some ways, the DNSSEC-aware mode integrated with the DNSSEC-aware mode can be considered as a new DNS transaction in case of detected attacks. That new DNS transaction first serves in the DNSSEC-aware mode for the DNS requests and then transitions to the DNSSEC-aware mode to generate replies.

**Integration of the two modes.** We present in detail how the two modes are integrated to defend against cache poisoning attacks. In particular, our example in Fig. 2 shows the defense procedure under the most mighty version of Kaminsky class attacks:

**(1)**: The attacker's client sends the target resolver a query for the IP address of "asq50pn.foo.com" below the target domain "foo.com". The domain "asq50pn.foo.com" is delicately crafted with random characters so that it is likely to miss the resolver's cache to trigger an outstanding query.

**(2a)**: The forgery authoritative server tries to send cache poisoning attempts to the target resolver guessing the transaction ID, etc. of the genius response until the failure responses accumulate to ToD. Each failure response may, e.g., guess a wrong transaction ID, and intends to inject the IP address of the forgery authoritative server, say "Y.Y.Y.Y".

**(2b)**: Roughly in parallel with (2a), the target resolver sends requests to the real authoritative name servers for "asq50pn.foo.com".

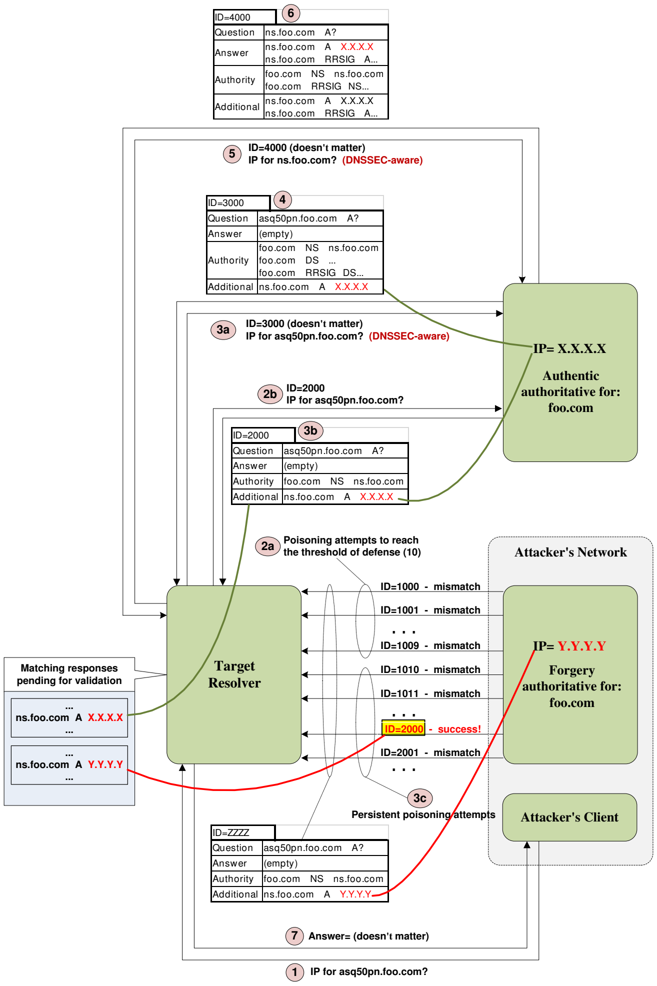
ID=4000
6
Question ns.foo.com A?
Answer ns.foo.com A X.X.X.X
ns.foo.com RRSIG A...
Authority foo.com NS ns.foo.com
foo.com RRSIG NS...
Additional ns.foo.com A X.X.X.X
ns.foo.com RRSIG A...
5
ID=4000 (doesn't matter)
IP for ns.foo.com? (DNSSEC-aware)
ID=3000
4
Question asq50pn.foo.com A?
Answer (empty)
Authority foo.com NS ns.foo.com
foo.com DS ...
foo.com RRSIG DS...
Additional ns.foo.com A X.X.X.X
3a
ID=3000 (doesn't matter)
IP for asq50pn.foo.com? (DNSSEC-aware)
IP= X.X.X.X
Authentic authoritative for: foo.com
2b
ID=2000
IP for asq50pn.foo.com?
ID=2000
3b
Question asq50pn.foo.com A?
Answer (empty)
Authority foo.com NS ns.foo.com
Additional ns.foo.com A X.X.X.X
2a
Poisoning attempts to reach the threshold of defense (10)
Attacker's Network
ID=1000 - mismatch
ID=1001 - mismatch
. . .
ID=1009 - mismatch
ID=1010 - mismatch
ID=1011 - mismatch
. . .
ID=2000 - success!
ID=2001 - mismatch
. . .
IP= Y.Y.Y.Y
Forgery authoritative for: foo.com
Matching responses pending for validation
...
ns.foo.com A X.X.X.X
...
...
ns.foo.com A Y.Y.Y.Y
...
Target Resolver
3c
Persistent poisoning attempts
Attacker's Client
ID=ZZZZ
Question asq50pn.foo.com A?
Answer (empty)
Authority foo.com NS ns.foo.com
Additional ns.foo.com A Y.Y.Y.Y
7
Answer= (doesn't matter)
1
IP for asq50pn.foo.com?

**Figure 2.** An example of the integration of the two modes.

**(3a)**: When the attack detection counts the number of failure responses to ToD, the target resolver starts the DNSSEC-aware mode by sending a DNSSEC-aware query for "asq50pn.foo.com" soliciting a validating response.

**(3b)**: Perhaps at the same time as (3a), the genius response arrives at the target resolver informing the IP address of the real authoritative server, say "X.X.X.X". However, as the DNSSEC-aware mode is already turned on, the response is hold on rather than simply accepted.

**(3c)**: The target resolver may still persistently be fed with cache poisoning responses after the ToD failure responses triggers the DNSSEC-aware mode. Before the validation response is returned, the continuous response guessing efforts do have a chance of success. The successful guessing response is also hold on for the future validation.

**(4)**: When the validating response is obtained by the target resolver, the relevant records in the validating response are subject to DNSSEC validation using the verified public key. That DNSSEC validation may render further DNSSEC transactions such as step (5) and (6) because some signatures (RRSIG records) over the interested data may be absent from the original validating response.

**(5)**: The target resolver initiates a new DNSSEC transaction to validate the IP address of the authoritative server ("ns.foo.com").

**(6)**: The new validating response contains a RRSIG record over the A type (IP address) record of "ns.foo.com". By then, the validating response can be validated.

**(7)**: By checking the hold-on list against the validating response, the IP address of "ns.foo.com", namely "X.X.X.X", is identified as genius and "Y.Y.Y.Y" as bogus. The validated record can thus be used by the target resolver in the final answer as well as in the cache.

## 3.3. Aggressive Use of Validating Response

**Efficiency and security concerns on the one-time use of validating response.** As discussed above, the validating response is introduced to defeat the cache poisoning attempt within the resolution transaction of a single query name. However, a validating response is underutilized if it is used only for a single query name, since some data may be shared among different query names. In the example of Fig. 2, the attacker may initiate a query for a new name other than "asq50pn.foo.com", say "b3rr5v.foo.com", immediately after he receives a genius response (indicating cache poisoning failure) rather than his intended bogus response (indicating cache poisoning success). Because the two query names fall into the same domain "foo.com", the data flows of the two defenses almost overlap except for where the query name "asq50pn.foo.com" is replaced with "b3rr5v.foo.com". But with the one-time use validating response, the resolver need at least two separate DNSSEC transactions for the two query names respectively. When successive cache poisoning attacks are launched using random generated query names within the same target domain, the resolver would waste a great number of DNSSEC transactions on the largely overlapping data. So the one-time use of validating response is sub-optimal in terms of efficiency.

Being a reasonably small value, ToD still allows for a minor enough chance of caching poisoning success within one window of the ODD transaction, since the defense leaves the initial ToD-1 caching poisoning attempts free of being detected and validated. While that threat is negligible for a short window, it may grow to serious when the brute force response guessing attack is rapid and continuous for a long window. This is because the success rate of caching poisoning increase dramatically with the number of cache poisoning attempts. The one-time use validating response only defeats the ToD th caching poisoning attempt and its successors within one ODD transaction, but it virtually does nothing to defend against the initial ToD-1 caching poisoning attempts in the next ODD transaction. So the one-time use of validating response weakens the ODD in terms of security.

Based on the above analysis, the underutilized validating response raises not only efficiency concerns but also security concerns. In order to maximize the utilization of validating response and minimize the cache poisoning opportunities, we propose to retain validating responses in cache for a long-lived defense rather than just use them once.

**Caching of validating response.** The signed records contained in the validating responses and validated by the recursive resolver should be regarded as more trustworthy than the unsigned records in the valid normal responses. Similar to the conventional DNS caching, those records are cached by the recursive resolver for a period to validate the normal responses. Nevertheless, the caching of validating responses differs from the conventional DNS caching in the following:

**a)** The validating records are given a priority over the unsigned normal records and they are stored in a priority cache other than a normal cache. Here "priority" means: a record in the priority cache can overwrite its unsigned counterparts in the normal cache if they conflict with each other; a record in the priority cache cannot be overwritten by any unsigned record in the more recent normal response; the life cycle of any record in the priority cache is ended either with its TTL expiration or with a replacement by a more recent validating response.

**b)** The records in the priority cache are basically used for validating normal responses. When a normal response arrives with any record conflicting with the priority cache, the recursive resolver should not accept the response. Instead it waits for its possible successor consistent with the priority cache until the resolution times out. The mechanism of waiting for genius response and denying bogus responses, used by the caching of validating responses, is very similar to that used by the fresh validating responses stated in the DNSSEC-aware mode. But for the sake of maintaining strong priority cache consistency, the recursive resolver should do more than simply return a timeout error as a response in case of resolution timeout.

**Proactive updating of validating response.** One common concern on the Time-to-Live (TTL) based caching such as DNS caching is the weak cache consistency. In DNS caching, a resolver stores a record in the cache as long as specified in that record's TTL field. The typical setting of TTL in DNS records ranges from 1 hour to 1 day. So the change of DNS records in authoritative servers is usually unlikely to be rapidly synchronized to resolvers because the resolvers follow the TTL expiration rule to invalidate the out-of-date cache entries and fetch the up-to-date copies upon requests. In conventional DNS specifications, cache inconsistency only poses a threat to the availability of Internet services because during the cache inconsistency period, the client served with out-to-date DNS records cannot reach the appropriate Internet servers. In the aggressive caching use of validating response, cache inconsistency, however, may result in a serious false positive of genuine response. This is simply because the out-to-date validating records in cache can deny genuine response containing more up-to-date copies of records. When both the genuine response and the bogus responses are invalidated by the stale validating records in cache, a resolution timeout takes place. So a resolution timeout may imply the possibility of cache inconsistency of validating records (and, of course, the possibility of authoritative server unresponsiveness or packet loss in the network).

Thus the hold-on mechanism specified in the DNSSEC-aware mode is slightly changed for caching of validating response. That is, the responses inconsistent with the responses in the priority cache are temporally hold on rather than discarded. Because the inconsistent responses may include the genuine response and the bogus responses in case of cache inconsistency of validating records, they are reserved for further validation.

To still obtain an up-to-date copy of validating record in cache when a resolution timeout (indicating the possibility of cache inconsistency), the resolver should proactively update the validating record in cache by acquiring a fresh validating response. The new validating response will has two usages: validating the hold-on responses and then returning the validated response if any; updating the corresponding validating records in cache. The responding process for the aggressive use of validating response is detailed in Fig. 3.

## 4. Analysis of Attack Surface

### 4.1. Attack surface if Turning off Aggressive Use of Validating Response

The window of opportunity is the time frame exploited by caching poisoning to inject bogus response into the recursive resolver. Within it, any bogus response matching the current outstanding query is accepted by the recursive resolver. With conventional DNS, the window of opportunity opens when the outstanding query departs from the recursive resolver and closes when the genuine response arrives at the recursive

```
$BogusCount \leftarrow 0$;
SEND THE REQUEST;
while $BogusCount < ToD$ do
    if $time\ out$ then
        RETURN(TIME_OUT); % The DNS resolution times out
    LISTEN TO THE RESPONSE;
    if $the\ response\ is\ bogus$ then
        $BogusCount \leftarrow BogusCount + 1$;
    else
        RETURN(THE RESPONSE); % The real response received
SEND THE DNSSEC REQUEST;
while $not\ time\ out$ do
    LISTEN TO THE RESPONSE;
    if $the\ response\ is\ validated$ then
        $ValidResponse \leftarrow the\ response$;
        USE $ValidResponse$ TO UPDATE $ValidCache$;
        if ($HoldonResponse \neq \phi$) & ($any\ response \in HoldonResponse = ValidResponse$) then
            RETURN(THE RESPONSE); % The real response received
    else if $the\ response\ is\ genuine$ then
        $HoldonResponse \leftarrow HoldonResponse \cup \{the\ response\}$;
    if ($ValidCache \neq \phi$) & ($HoldonResponse \neq \phi$) then
        if $any\ response \in HoldonResponse\ matches\ ValidCache$ then
            RETURN(THE RESPONSE); % The real response received
RETURN(TIME_OUT); % The DNS resolution times out
```

**Figure 3.** The responding process of DNSSEC-aware mode with aggressive use of validating response.

resolver. That is, the window of opportunity in conventional DNS is equivalent to the response time at the recursive resolver. For a given window of opportunity, attackers typically hasten its cache poisoning attempts in order to employ more bogus responses for a more chance of success.

ODD limits the window of opportunity in the number of bogus responses available for cache poisoning rather than in the length of time. Only ToD bogus responses at maximum are permitted for a successful cache poisoning attempt. For attackers with an average ability of sending bogus responses, the window of opportunity allowed by conventional DNS is far longer than that allowed by ODD if ToD is set as a sound value ranging from 1 to 10.

Competent as it is in terms of window of opportunity, ODD still has some, if non-negligible for some cases, impacts on the overall response time. Because an extra validating request is initiated during a transition to DNSSEC-aware mode, the recursive resolver in ODD has to await the validating response even after the genuine response is received. The recursive resolver should hold on any matching response since ToD is reached (and a validating request is initiated) until the validating response arrives. The validating response is then used to validate all matching responses on hold.

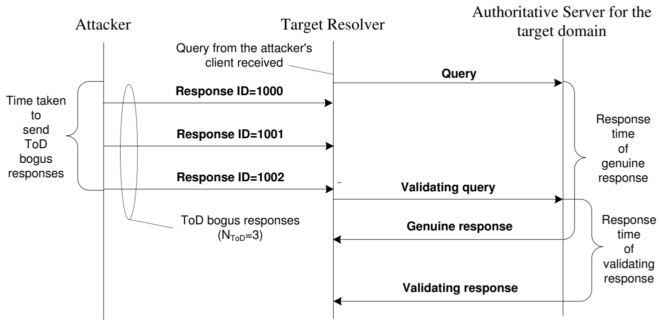
Attacker
Target Resolver
Authoritative Server for the target domain
Query from the attacker's client received
Query
Time taken to send ToD bogus responses
Response ID=1000
Response ID=1001
Response ID=1002
Response time of genuine response
Validating query
ToD bogus responses ($N_{ToD}$=3)
Genuine response
Response time of validating response
Validating response

**Figure 4.** Illustration of a cache poisoning attempt.

The final response is returned as a successful genuine response only if at least one matching response is validated by the validating response. Otherwise, if no matching response is ever received, or any matching response received is ever validated as bogus, or no validating response is ever received, a final response of timeout is returned. In some (relatively unlikely) cases, it is also possible that the genuine response arrives later than the validating response. If so, the recursive resolver with a validating response may still expect a matching response to be validated as the received matching response, if any, is likely to be validated as bogus.

Consider the most common case that the genuine response as well as the validating response is successfully received by the recursive resolver. Without loss of generality, the overall response time of ODD is jointly determined by the response time of genuine response and the sum of time taken to count ToD failure responses and the response time of validating response. As illustrated in Fig. 4, it can be expressed as:

$$R_T = \max\{T_{ToD} + R_v,\ R_g\} \tag{1}$$

Where $T_{ToD}$ is the time taken to reach ToD failure responses, $R_v$ is the response time of validating response, and $R_g$ is the response time of genuine response.

As a maximum-efficacy strategy, the attacker would tend to persistently send response guessing packets until he gets to know the compromise effort is a success or not. The only way for an attacker to become aware about the result of cache poisoning is to check the final responses to his initial queries. If the response contains the bogus information that the attacker has tried to inject, the cache poisoning is successful. Otherwise, if the response is left unpolluted by the bogus information, the cache poisoning is a failure. The effective cache poisoning attempts are virtually ended at the time when the DNSSEC-aware mode is triggered and therefore all responses are hold on, because the lately retrieved DNSSEC response will invalidate all matching but bogus response injected by the attacker. However, the attacker, who is likely to be still unaware about this, would waste his resources to keep trying cache poisoning attempts until the final response is returned from the target recursive resolver. Once the attacker finds a failure of cache poisoning in the response, the best strategy for him is to stop sending bogus responses for his recent queried domain name but initiate a new query for a new domain name below the target domain. That is virtually the end of the old round of cache poisoning attempt and the start of a new old round of cache poisoning attempt. That periodical cache poisoning attempt may repeat likewise until a success cache poisoning is found in the response by the target resolver.

Despite that the response time of ODD is somewhat comparable to that of conventional DNS, the window of opportunity within which effective response guessing packets are tried is remarkably reduced by ODD. We can find that the period of each round of cache poisoning attempt is approximated as the response time for both conventional DNS and ODD. So in comparison with conventional cache poisoning, the capability of cache poisoning attacks is significantly limited by ODD because of the diminished window of opportunity in the period of each round of cache poisoning attempt.

Let the bandwidth available for cache poisoning be $B$, and the average size of the bogus responses be $S$. Then the maximum rate of bogus response is $B/S$. The valid time exploited for cache poisoning in a period of each round of cache poisoning attempt is decreased to the time taken to send ToD bogus responses. Let the value of ToD be $N_{ToD}$. So the utilization of cache poisoning is

$$U = \frac{(N_{ToD} - 1)S}{R_T B} \tag{2}$$

E.q. 2 indicates that an increase of $N_{ToD}$ or a decrease of $R_T$ will favor the utilization of cache poisoning. But E.q. 1 also gives rise to an illusion that a smaller bandwidth of an attacker would benefit cache poisoning in the way that it improves the utilization of cache poisoning. So in E.q. 3 we define an alternative metric to express the equivalent rate of cache poisoning attempts using ODD. E.q. 3 shows that the equivalent rate of injecting cache poisoning responses is virtually dependent on $N_{ToD}$ and $R_T$ rather than the attacker's sending capability $B/S$.

$$V_E = \frac{N_{ToD} - 1}{R_T} \tag{3}$$

In a period of each round of cache poisoning attempt, the attacker actually has an opportunity of $N_{ToD} - 1$ cache poisoning responses to guess the genuine response by matching the current outstanding queries.

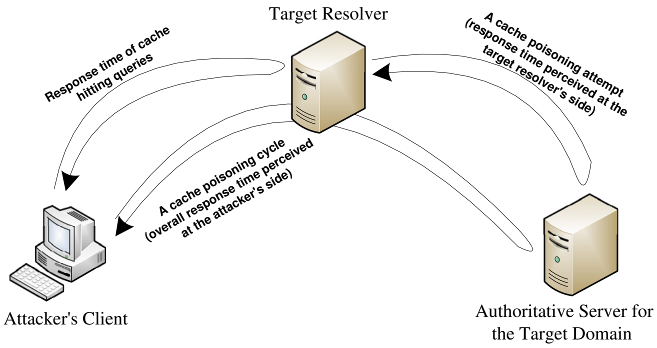
Target Resolver
Response time of cache hitting queries
A cache poisoning attempt (response time perceived at the target resolver's side)
A cache poisoning cycle (overall response time perceived at the attacker's side)
Attacker's Client
Authoritative Server for the Target Domain

**Figure 5.** Illustration of a cache poisoning cycle.

A cache poisoning cycle is defined as a non-stop action that can be repeated continuously for the sake of a quickest compromise. As illustrated in Fig. 5, the period of a cache poisoning cycle (namely overall response time perceived at the attacker's side) roughly consists of response time perceived at the target resolver's side (namely the period of a cache poisoning attempt $R_T$) and RTT between the attacker's client and the target resolver (namely response time of cache hitting queries). The latter refers to the delay taken to deliver the query from the attacker's client to the target resolver and return its response if the query is hit by the target resolver's cache. For each cache poisoning cycle, only $N_{ToD} - 1$ cache poisoning responses at maximum are allowed at the target resolver no matter how many cache poisoning responses are virtually generated by the attacker. The period of a cache poisoning cycle is

$$R_C = R_H + R_T \tag{4}$$

## 4.2. Attack Surface if Turning on Aggressive Use of Validating Response

As analyzed above, the attacker knows nothing about whether his cache poisoning attempts success until the final response is forwarded to it from the target resolver. If the final response is consistent with the bogus response it has tried to inject, the target resolver knows that cache poisoning is done. Otherwise, the attacker understands that its cache poisoning efforts for that window of opportunity are fruitless in this cache poisoning cycle, and then he can initiate a new cache poisoning cycle by generating a request for a different random name in the target domain. As the best efforts to compromise the target resolver as soon as possible, the attacker is likely to persistently send bogus responses to the target resolver at a rate of his utmost during a cache poisoning cycle. Among those bogus responses, only $N_{ToD} - 1$ attempts are useful for cache poisoning if they fail to hit. Therefore, a lot of caching poisoning attempts are expected to be a waste of resources of attacker. Even so, the best strategy of attack is still to feed a flurry of bogus responses as fast as possible because this may reduce the time taken to reach a given ToD, namely a given number of valid cache poisoning attempts. In this way, the size of the window of opportunity is also decreased. The attacker can thereby continue its efforts by starting a new cache poisoning cycle as early as possible.

It is a rather rare chance for cache poisoning attacker to guess the genuine response successfully in $N_{ToD}-1$ attempts if $N_{ToD}$ is set as a sound small value. However, a sensible attacker can perform every $N_{ToD}-1$ cache poisoning responses in successive cache poisoning cycles. A minor probability of compromise in one cache poisoning cycle may add up to a non-negligible probability over time and over cache poisoning cycles.

Aggressive use of validating response may greatly strengthen the defense capability of ODD by prolonging the cache poisoning cycle with a constant window of opportunity. Let the resident time of validated record of the validating response in the cache be $T_r$. The protection provide by the validating response lasts $T_r$ after each cache poisoning cycle. Taking that prolonged protection into account, the utilization in E.q. 2 becomes

$$U' = \frac{(N_{ToD}-1)S}{(R_T+T_r)B} \tag{5}$$

The equivalent rate of injecting cache poisoning responses E.q. 3 can be rewritten as

$$V'_E = \frac{N_{ToD}-1}{R_T+T_r} \tag{6}$$

The period of a cache poisoning cycle is

$$R'_C = R_H + R_T + T_r \tag{7}$$

Most TTLs of the authoritative records, especially those of the authoritative servers' records, range from minutes to days. And the update intervals of those records are usually expected to be larger or much larger than their TTLs. That setting makes sense in that the records in cache are fairly unlikely to be inconsistent with the authoritative ones. More frequently updated records are thus more frequently synchronized from their authoritative servers to recursive resolver's cache because of their relatively smaller TTLs.

Given that TTL setting convention, there is a chance, albeit usually a small one, that the data in cache is outdated before its TTL expires. Some update may be taken on an authoritative record and meanwhile its counterpart in cache is unaware of the update because it is kept as it is in cache until its TTL expires. When that happens to a priority cache, the proposed scheme still has ways to handle it. As stated above, the outdated data in a priority cache tends to deny both the bogus responses and the up-to-date response because they are both inconsistent with the priority cache. Since all responses are discarded as bogus, the resolver will persist the hold-on until response timeout occurs. Then the old data in the priority cache is likely to be updated by the new validating response containing the up-to-date data. Another possibility of response timeout is that the genuine response is, rather than updated, lost in the path or the authoritative server is unresponsive. In that case of response absence, the old data in the priority cache may be simply renewed by a new one as its residual TTL is replaced with a full TTL. The detailed analysis of $T_r$ is presented in Section 5.1.

## 5. Performance Analysis

### 5.1. Query Load on the Authoritative Server

ODD never initiates DNSSEC transactions unless possible cache poisoning attack is detected at the target resolver. Thus for a vast majority of recursive resolvers which are not constantly targeted by cache poisoning adversaries, ODD is lightweight in the name resolution cost at both recursive resolvers and authoritative servers because DNSSEC is much less used by ODD than by the existing DNSSEC deployment strategy.

Consider the case of most severe cache poisoning attack targeting the victim resolver. That is, the attacker continuously sends caching poisoning responses at a high rate towards the target resolver. A DNSSEC transaction is generated by the target resolver if and only if:

- The validated records in the priority cache expire so that an immediate flurry of caching poisoning responses triggers the DNSSEC-aware mode;
- No validated response is found until timeout because of the updated authoritative records. As DNSSEC is triggered roughly either by the expiration of TTL or by the updated authoritative records, we first investigate the event of queries triggered by them separately. Without loss of generality, we assume the TTL follows a probability distribution function.

If the target record is heavily requested, the times between successive events (queries) can be approximated by the value of TTL at the instances of events. Let the TTLs or the successive inter-event times are independently and identically distributed. So there is a renewal process in operation for the TTL-triggered queries. Assume that the successive times between the updates of authoritative records are independently and identically distributed. So there is also a renewal process in operation for the update-triggered queries.

However, it is not true that the two renewal processes can be supposed to be independent renewal processes in operation simultaneously. No matter how long the TTL elapses, the update-triggered queries take place merely following the inter-update times. This means the renewal process of update-triggered queries is independent of the renewal process of TTL-triggered queries. But the inter-event times of TTL-triggered queries are dependent of those of update-triggered queries. For example, if there is no update between two successive TTL-triggered queries, their inter-time is a TTL; if there is one update between them, the residual TTL is renewed to a full TTL at the instance of update, and so their inter-time is prolonged to be a full TTL plus a residual TTL; if there is more than one updates between them, the residual TTL is renewed more than one times, and their inter-time becomes a full TTL plus more than one residual TTLs. Given the dependence analyzed above, the sequence of events of DNSSEC queries cannot be considered to be formed by superposing the two individual processes. Instead, we depict the process of DNSSEC queries using the code in Fig. 6.

```
% The present time is initialized at an instance of update-triggered query
$t \leftarrow 0$; % The time is initialized as zero
$T \leftarrow TTL$; % The residual TTL is a TTL after an update-triggered query
while True do
    if $T = 0$ then
        SEND A REQUEST; % Initiate a TTL-triggered query
        $T \leftarrow TTL$;
    else if an authoritative update occurs at $t$ then
        SEND A REQUEST; % Initiate an update-triggered query
        $T \leftarrow TTL$;
    $t \leftarrow ELAPSE(t)$; % Time elapses
    $T \leftarrow T - (ELAPSE(T) - T)$; % The residual TTL decreases as time elapses
```

**Figure 6.** The process of DNSSEC query event by ODD.

## 5.2. Cache Poisoning Success Rate

In conventional Kaminsky cache poisoning attacks, the attacker can balance between the number of outstanding requests and the number of bogus response attempts at will to achieve maximum efficiency. Because the number of bogus response attempts is limited for cache poisoning attacks if protected by the proposed scheme, the attacker has to create more duplicate requests for the same target domains subject to bogus response attempts in a bid to increase the probability of successful compromise. However, the number of outstanding requests are also bounded by two aspects in practice:

- The maximum number of outstanding requests is usually set as a default configuration in most widely used authoritative server implementations. Authoritative servers will thereby discard excessive outstanding requests surpassing the configured limit. So any efforts of producing over-the-limit outstanding requests will prove fruitless.

- The window allowed to persistently elicit outstanding requests may be bounded by the response time. When the resolver begins receiving a response matching an outstanding request in the wait-for-response list, the list will not necessarily be on the rise since then because the responding rate may be not below the request rate. Hence a conservative estimation of the window of outstanding requests is the response time perceived by the target resolver. As an equivalent of the first limit, the window can be converted to the number of outstanding requests if the sending rate is constant.

In summary, the maximum number of outstanding requests is the minimum of the two limits stated above.

Let the threshold of bogus response attempts be $H$, and the maximum number of outstanding requests be $D$. We can express the cumulative probability of cache poisoning failure in all attempts up to and including the $H$ th attempt as

$$P_D(H) = P(\textit{the } 1\textit{st attempt misses, the } 2\textit{nd attempt misses, ..., the } H \textit{ th attempt misses} \mid D \textit{ identical outstanding queries}) \tag{8}$$

If $H \ll (I + P) * N$, $P_D(H)$ can be written as

$$P_D(H) = (1 - D/((I + P) * N))^H \tag{9}$$

If no window extension mechanism is applied, the window allocated for launching the $H$ th attempts equals the window of eliciting outstanding requests plus the window of validating the responses. As analyzed above, the first is roughly the response time and the second is also approximated as the response time. That is, one round of cache poisoning attempt takes two response times to obtain a success rate of $1 - P_D(H)$. The success rate of cache poisoning within $i$ rounds of cache poisoning attempt is $1 - P_D(H)^i$.

The window extension mechanism will dramatically diminish the success rate of cache poisoning in a given time because one round of cache poisoning attempt with a constant success rate of cache poisoning just lasts much longer. The cached validating records suppress the attacker from initiating a new round of cache poisoning attempt immediately after the old round proves a failure. A new window starts whenever the validating records expires from the cache. So the length of window is at least the TTL of the validating records. Furthermore, the window may be prolonged to above a TTL if updates take place before the TTL expires. In such cases, the continuous elapse of TTL is interrupted by any update which renews the residual TTL to a full TTL. Then the cached validating records will still at least last a full TTL to expires its TTL. During this period, the residual TTL may be renewed again and again whenever a update occurs before it reaches zero. In general, the effects of window extension are better pronounced for a more frequent update.

## 6. Model Checking Results

Probabilistic model checking is one of the most common used formal verification technique for the modeling and analysis of stochastic systems. In probabilistic model checking, the construction and analysis of a probabilistic model explores all possible all possible states that can occur as well as all possible process scheduling. In comparison with discrete-event simulation techniques, which generate approximate results based on averaging results from a large number of random samples, probabilistic model checking typically yields exact results by employing numerical computation in an efficient and exhaustive fashion. Thus probabilistic model checking is applied to the design and analysis of complex systems across a broad spectrum of application domains.

PRISM [17] is an open-source probabilistic model checker, providing support for building and analyzing several types of probabilistic models: discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs), Markov decision processes (MDPs), etc. plus extensions of these models with costs and rewards. A probabilistic model in PRISM is constructed as a set of state-transition actions (called as commands) in each module (each module comprises a set of commands and a set of variables): a guard is associated with each command, representing a predicate over the command; a command updates some variables with new values if the guard is satisfied; the probability (in the case of DTMCs) or rate (in the case of CTMCs) of the transition of each variable in a command is assigned; the label for each command allows commands in different modules to share their labels as a mechanism for them to synchronize in a rate as the product of the rates of the individual transitions.

### 6.1. Modeling Cache Poisoning Attack & ODD

We model Kaminsky cache poisoning attack as a CTMC using PRISM. In modeling the attack, we assume that in each round of cache poisoning attempt, the queries originated from the attackerąŕs client look up a random generated domain such that they will never hit the target resolver's cache. Instead those queries will trigger the target resolver to issue their corresponding requests towards the authoritative servers. Attackers are very likely to adopt this strategy in order to strengthen their attack efficacy. We also assume that the IP addresses of the target domain's authoritative servers are always maintained in the cache of the target resolver such as queries in each round of cache poisoning attempt will never render the target resolver request for the IP addresses of the target domain's authoritative servers. The assumption is valid in the sense that it is a very rare case that the target domain in question finds no cached records of its authoritative servers in a cache poisoning attempt. The TTLs of records of authoritative servers typically range from hours to days or even months for the sake of lowering the wide-area DNS traffic as well as the authoritative server's load. In comparison, one round of cache poisoning attempt lasts for a period comparable to a DNS lookup RTT (usually from tens to hundreds of milliseconds). So the simplification is sound because of the significantly minor percentage of authoritative server's cache miss for a cache poisoning attempt.

Our model defines the following four modules, two for the attacker side and two for the victim side:

**Attacker's Client (AC):** AC is the DNS client in the control of the attacker. Like normal DNS clients, it functions as an originator of DNS requests. However, its requests aim at opening a window of opportunity for bogus responses rather than getting the real responses. The other exploiting of AC is to notify the attacker of the failure of a cache poisoning attempt in operation upon the receipt of the real responses to all requests. Then the attacker may make AC initiate a new round of cache poisoning attempt by querying a new random generated domain.

**Attacker's Bogus Authoritative Server (AS):** AS is the bogus DNS authoritative server in the control of the attacker. AS is coordinated with AC to launch a cache poisoning attempt. When AC has issued DNS requests for the target domain, AS starts feeding the target resolver with bogus responses. When AC has received the real responses to all requests (this is the end of cache poisoning attempt in operation), AS stops sending bogus responses to the old domain and awaits sending bogus responses to the new domain in the next round of cache poisoning attempt.

**Target resolver (TR):** TR is the victim of the cache poisoning attack. When it receives a request from AC, it will forward the request to the real authoritative server. Then the bogus response may be potentially accepted by TR with a probability. If a bogus response fails to hit any of the outstanding requests, it is counted into the overall number of failed bogus responses by TR. When the number reaches the threshold, a validating request is initiated, and all received responses to the original requests are hold on from then on until the corresponding response to the validating request arrives. That corresponding response will then validate all responses which are finally returned to AC.

**Real Authoritative Server (RS):** RS is authoritative for the target domain. It operates as no other than a normal DNS authoritative server. That is, no matter the incoming request is a normal request or a validating request, RS returns a DNS-compliant response.

The model parameters are the following:

Sending rate of queries from AC to TR $R_a$: The rate of queries that AC sends to TR in order to create the wait-to-response requests.

Number of outstanding queries $N$: The number of queries for the same domains AC sends to TR in each round of cache poisoning attempt.

Guess rate of bogus responses $R_g$: The rate of the bogus responses that AS sends to TR in order to match one of the wait-to-response requests.

Sending rate of queries from TR to RS $R_t$.

Responding rate of queries from RS to TR $R_r$.

These two parameters above jointly determines the length of window of opportunity: $L_w = 1/R_t + 1/R_r$.

The total number of query ID and transaction ID space to be guesses by the cache poisoning attempt $S$.

Each module defines certain actions, which synchronize with appropriate actions from other modules. Since our model is a CTMC, each action (CTMC transition) has an associated rate. Actions also have associated preconditions that need to be satisfied for their execution to take place. We now describe some of the important actions for each module. Unless stated otherwise, each action is executed with a constant rate of 1.

**Actions Defined for AC:**

[Send request to TR] With rate request_rate, AC sends requests to TR. When the number of queries accumulates to number of outstanding queries in each round of cache poisoning attempt, AC awaits for the next round of cache poisoning attempt to implement this action again. This action is synchronized with action [Send request to TR] of TR.

[Receive response from TR] AC receives response from TR. When the number of received responses accumulates to number of outstanding queries in each round of cache poisoning attempt, AC awaits for the next round of cache poisoning attempt to implement this action again.

[Initialize a new round of cache poisoning attempt] AC initializes a new round of cache poisoning attempt at the time the number of received responses accumulates to number of outstanding queries. The number of queries and the number of received responses are both initialized as zero.

**Actions Defined for AS:**

[Send a bogus response of correct guess to TR] With rate guess_rate, AS sends a bogus response of correct guess to TR. The action is allowed when the number of queries accumulates to number of outstanding queries. This action is synchronized with action [Send a bogus response of correct guess to TR] of TR.

[Send a bogus response of incorrect guess to TR] With rate guess_rate, AS sends a bogus response of incorrect guess to TR. The action is allowed when the number of queries accumulates to number of outstanding queries. This action is synchronized with action [Send a bogus response of incorrect guess to TR] of TR.

**Actions Defined for TR:**

[Send request to TR] TR receives a request from AC and places it in the queue of wait-for-response. This action is synchronized with action [Send request to TR] of AC.

[Send request to RS] TR sends a request to RS if the queue of wait-for-response is not empty. The action opens a window of opportunity by setting a switch variable of guessing. That is, action [Send a bogus response of correct guess to TR] and action [Send a bogus response of incorrect guess to TR] can be enacted following this action. This action is synchronized with action [Send request to RS] of RS.

[Receive response from RS] TR receives a response from RS if the queue of wait-for-response is not empty and the cache poisoning attack does not succeed yet. This action pops a request from the queue of wait-for-response and places it in the queue of wait-to-response. This action is synchronized with action [Receive response from RS] of RS.

[Receive response from TR] TR finally returns a response to AC if the following are satisfied: the queue of wait-to-response is not empty; the number of incorrect guesses is below the threshold or the response is validated when the number of incorrect guesses reaches the threshold. This action pops the queue of wait-to-response. Furthermore, if the number of responses reaches the number of requests, all relevant parameters are initialized to meet the next round of cache poisoning attempt.

[Send a bogus response of correct guess to TR] AS sends a bogus response of correct guess to TR if the switch variable of guessing is set, the cache poisoning attack does not succeed yet, and the number of incorrect guesses is below the threshold. The rate of this action is dependent of the probability of cache poisoning success, which is expressed as: $trials/(query\_id * port\_id - space\_explored)$. This action is synchronized with action [Send a bogus response of correct guess to TR] of AS.

[Send a bogus response of incorrect guess to TR] AS sends a bogus response of incorrect guess to TR if the switch variable of guessing is set, the cache poisoning attack does not succeed yet, and the number of incorrect guesses is below the threshold. The rate of this action is dependent of the probability of cache poisoning failure, which is expressed as: $1 - trials/(query\_id * port\_id - space\_explored)$. This action is synchronized with action [Send a bogus response of incorrect guess to TR] of AS.

[Send a validating request to RS] AS sends a validating request to RS if the number of incorrect guesses reaches the threshold. This action is synchronized with action [Send a validating request to RS] of RS.

[Receive a validating response from RS] AS receives a validating response from RS if there is an outstanding validating request. This action is synchronized with action [Receive a validating response from RS] of RS.

**Actions Defined for RS:**

[Send request to RS] With rate request_arrive_rate, RS receives a request from TR. This action is synchronized with action [Send request to RS] of TR.

[Receive response from RS] With rate response_serve_rate, RS processes a request to generate a response returned to TR. This action is synchronized with action [Receive response from RS] of TR.

[Send a validating request to RS] With rate validating_request_arrive_rate, RS receives a validating request from TR. This action is synchronized with action [Send a validating request to RS] of TR.

[Receive a validating response from RS] With rate validating_response_serve_rate, RS processes a request to generate a validating response returned to TR. This action is synchronized with action [Receive a validating response from RS] of TR.

## 6.2. Results of Query Load

To investigate the effects of combination of TTL expiration and authoritative update on the intertime of DNSSEC queries, we generate a sequence of authoritative update events following a probabilistic distribution while setting the TTLs in the DNSSEC responses as constant and probabilistic values respectively. In one set of experiments, we let the TTLs of records in question be evenly distributed on the interval from 500s to 1500s. In the other set of experiments, the TTLs of records in question take a constant value as 1000s. For both sets of experiments, the intertime of authoritative updates follows exponential distribution with the parameter ranging from 100s to 1400s.

We use Monte Carlo method to estimate the mean of intertimes of DNSSEC queries. In each experiment, 100,000 times of authoritative updates are generated from an exponential distribution. A number of TTLs, taking either constant values or probabilistic values, are also produced to cover the same time span at the instances when the predecessor TTL expires or authoritative update takes place.

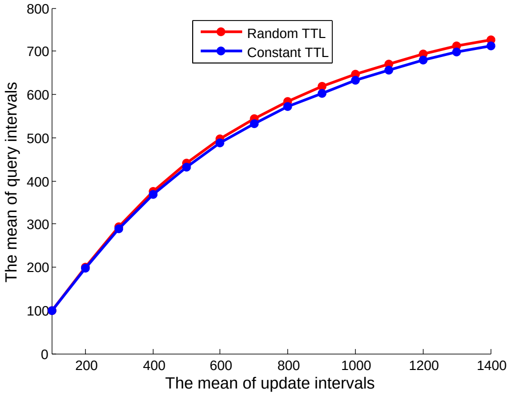

**Figure 7.** DNSSEC query intervals vs authoritative update intervals.

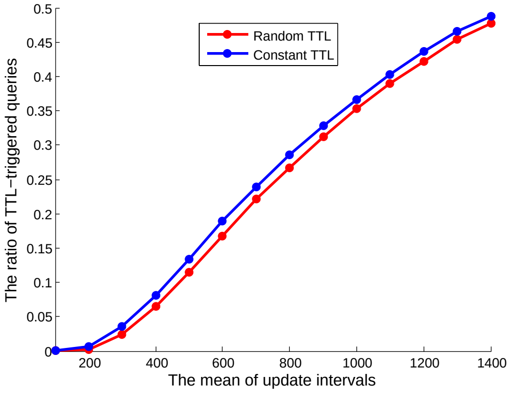

**Figure 8.** Ratio of TTL–triggered queries vs authoritative update intervals.

Fig. 7 illustrates how DNSSEC query intervals change with authoritative update intervals. We can see that a very small authoritative update interval has almost the same DNSSEC query interval because TTL expiration rarely happens. But for a larger authoritative update interval, the limiting effect of TTL is better pronounced because a TTL has more chance of being smaller than an authoritative update interval thus more chance of expiration. Random TTLs, though have the same mean as constant TTLs, tend to cause a slightly larger DNSSEC query intervals and thereby a smaller DNSSEC

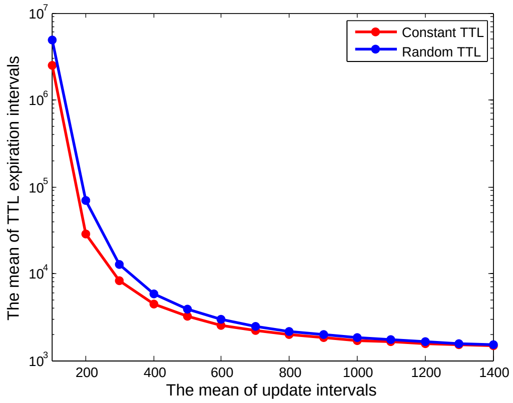

**Figure 9.** TTL expiration intervals vs authoritative update intervals.

query load on authoritative servers. The ratio of TTL-triggered queries is illustrated in Fig. 8. We can see that the ratio of TTL-triggered queries grows as the mean of update intervals increases. But the authoritative update tends to pronounce more than TTL expiration on triggering DNSSEC queries even if they share the same mean interval. As shown in Fig. 9, when both update interval and TTL take a mean of 1000s, TTL-triggered DNSSEC queries only account for about 36% of the total. That can be explained by the fact that the event of authoritative update is independent of and never superceded by the event of TTL expiration while the even arrival of TTL expiration may be interrupted and restarted by an authoritative update.

It is obvious that DNSSEC query interval will be larger if authoritative update and TTL expiration are independent. So in order to examine the lower bound of DNSSEC query interval or the upper bound of DNSSEC query rate, we assume that authoritative update and TTL expiration are independent. Then the mean DNSSEC query interval can be written as

$$I_{overall} = \frac{I_{update} * I_{ttl}}{I_{update} + I_{ttl}} \qquad (10)$$

Where $I_{update}$ and $I_{ttl}$ represent the authoritative update interval and the TTL respectively.

As can be seen from Fig. 7 and Fig. 8, we can conclude that the maximum DNSSEC query rate of ODD under intense cache poisoning attempts is of the same order as the minimum of the authoritative record update rate and the reciprocal of TTL.

## 6.3. Results of Cache Poisoning Success Rate

We configure the default values in Tab. 1 for the parameters in the model checking unless their values are otherwise stated.

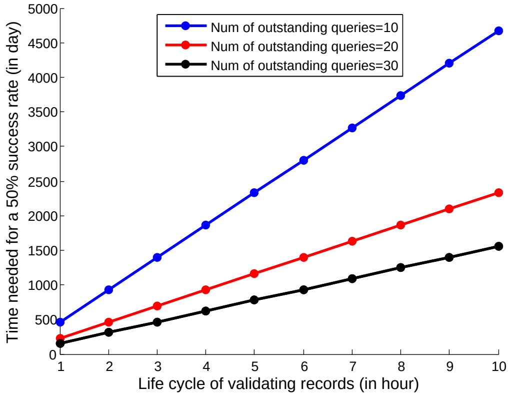

**Figure 10.** Time needed for a 50% success rate vs life cycle of validating records (ToD=3).

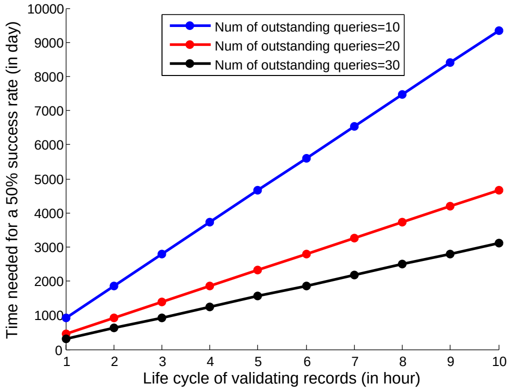

**Figure 11.** Time needed for a 50% success rate vs life cycle of validating records (ToD=2).

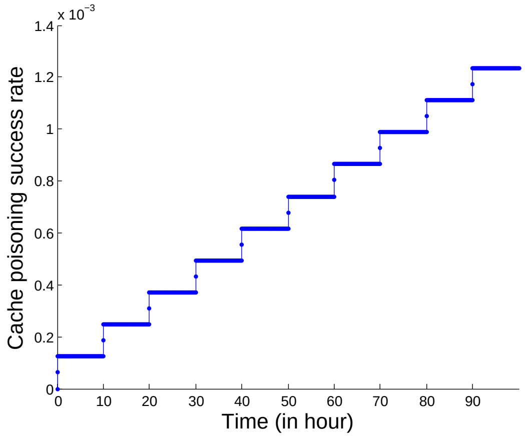

**Figure 12.** Cache poisoning success rate vs time (ToD=3).

First, we illustrate the time needed for a 50% success rate under different life cycle of validating records in Fig. 10 (ToD=3). We can see that the time cost of cache poisoning roughly grows linearly with the life cycle of validating records. For a life cycle above 10 hours, the time required for a 50% success rate

**Table 1.** Parameters and their settings.

| Parameter | Setting |
|---|---|
| Number distinct IDs available | 65536 |
| Number of ports used (ports less than 1024 are unavailable) | 64000 |
| Number of authoritative servers for a domain | 2.5 |
| Window of opportunity | 0.02 s |
| Number of identical outstanding queries of a resolver | 20 |
| Query sending rate from the target resolver to the authoritative servers | 100 qps |
| Query responding rate from the authoritative servers to the target resolver | 100 qps |
| Query sending rate from the attacker to the target resolver to create outstanding requests | 1000 qps |
| ToD | 3 |
| Bogus responding rate from the attacker to the target resolver | 100 |
| Life cycle of validating records | 10 hour |

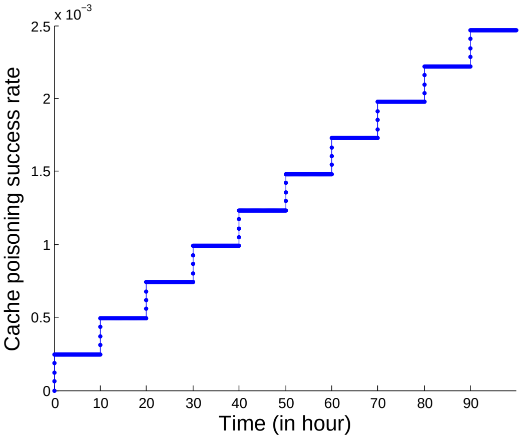

**Figure 13.** Cache poisoning success rate vs time (ToD=5).

amounts to no less than 2 years. This is because the longer are the validating records provided in cache to defend against cache poisoning attacks, the longer does an attacker have to wait to embark the next cache poisoning attempt (if the current attempt fails). As the TTLs of many authoritative records are set in the order of days or even weeks, it is very hard in practice to compromise them through cache poisoning attacks. Fig. 10 also shows creating more outstanding queries may dramatically decrease the difficulty of cache poisoning. Thus in the defense, the resolver should not allow excessive identical outstanding queries in order to prevent an unacceptable success rate of cache poisoning.

Second, we investigate the impacts of ToD on the success rate. In Fig. 11, the time needed for a 50% success rate is shown when the ToD is lowered to 2. We can see that limiting ToD helps significantly to suppress the success rate of cache poisoning. Since ToD defines the maximum number of forgery responses (ToD-1) allowed without defense, a larger ToD means more chance of guessing attempts in a cache poisoning attempt thus a larger success rate. To ensure the efficacy of ODD, ToD should be set as a sound small value.

Third, we study how the cache poisoning success rate evolves over time. In Fig. 12, we can see that the success rate over time grows like a stair-step shape. In the curve, each step virtually represents a cache poisoning attempt in time and an accumulation of ToD-1 forgery responses in success rate. And the width of each stair-step is dominated by the life cycle of validating response. When ToD is three in Fig. 12, there are two forgery responses aggregated in a cache poisoning attempt to increase the overall success rate.

Fourth, how the setting of ToD impacts the cache poisoning success rate is studied. As illustrated in Fig. 13, the increase of ToD from 3 to 5 will lessen the defense of ODD against cache poisoning attacks. While the width of each stair-step stays the same as Fig. 12, the jump of each stair-step in the success rate is doubled. So the overall success rate grows much faster than Fig. 12. This shows again that a large ToD may undermine the defense capability of ODD.

## 7. Conclusions

DNSSEC deployment suffers from its significant costs which in turn slow its progess. The resulting long transition to DNSSEC leaves a large name space still vulnerable to cache poisoning attacks. To speed up DNSSEC adoption and thereby narrow the window of transitional risks, a lightweight DNSSEC solution was proposed. The attack detection performed by recursive resolvers is employed to take up DNSSEC on demand rather than incessantly. The lightly used DNSSEC not only greatly lowers the DNSSEC overheads but also basically reserves the DNSSEC defense capability against cache poisoning attacks.

## References

[1] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, Resource Records for the DNS Security Extensions, RFC 4034, Mar. 2005.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose, Protocol Modifications for the DNS Security Extensions, RFC 4035, Mar. 2005.

[3] L. Yuan, C. C. Chen, P. Mohapatra, C. N. Chuah, and K. Kant, A Proxy View of Quality of Domain Name Service, Poisoning Attacks and Survival Strategies, ACM Trans. Internet Technol, 12(3), Article 9, 26 pages, 2013.

[4] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee, Increased DNS Forgery Resistance Through 0x20-bit Encoding: Security via Leet Queries, Proc. of the 15th ACM conference on Computer and communications security (CCS '08), 211-222, 2018.

[5] D. Kaminsky, It's the End of the Cache As We Know It, BlackHat 2008.

[6] G. Huston, G. Michaelson, Measuring DNSSEC Performance, 2013. potaroo.net/ispcol/2013-05/dnssec-performance.pdf

[7] D. Migault, C.Girard, and M. Laurent, A Performance View on DNSSEC Migration, Proc. of the Int. Conf. on Network and Service Management (CNSM'10), 469-474, 2010.

[8] B. Ager, H. Dreger, and A. Feldmann, Predicting the DNSSEC Overhead Using DNS Traces, Proc. of the Conf. on Information Sciences and Systems (CISS'06), 1484-1489, 2006.

[9] W. Lian, E. Rescorla, H. Shacham, and S. Savage, Measuring the Practical Impact of DNSSEC Deployment, Proc. of the USENIX SEC'13, 573-588, 2013.

[10] ICANN, TLD DNSSEC Report (2018-04-23 00:02:21), 2018. stats.research.icann.org/dns/tld_report/

[11] L. Fan, Y. Wang, X. Cheng, and J. Li, Prevent DNS Cache Poisoning Using Security Proxy, Proc. of the Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT'11), 387-393, 2011.

[12] K. Schomp, M. Allman, and M. Rabinovich, DNS Resolvers Considered Harmful, Proc. of the ACM HotNets'14, 16-22, 2014.

[13] H. M. Sun, W. H. Chang, S. Y. Chang, and Y. H. Lin, DepenDNS: Dependable Mechanism Against DNS Cache Poisoning, Proc. of the Int. Conf. on Cryptology and Network Security (CANS'09), 174-188, 2009.

[14] H. Shulman and M. Waidner, Towards Forensic Analysis of Attacks With DNSSEC, Proc. of the IEEE Security and Privacy Workshops (SPW'14), 69-76, 2014.

[15] Z. Wang, POSTER: On the Capability of DNS Cache Poisoning Attacks, Proc. of the ACM CCS'14, 1523-1525, 2014.

[16] Z. Wang, A Revisit of DNS Kaminsky Cache Poisoning Attacks, Proc. of the IEEE GLOBECOM'15, 1-6, 2015.

[17] M. Kwiatkowska, G. Norman, and D. Parker, PRISM 4.0: Verification of Probabilistic Real-time Systems, Proc. of the Int. Conf. on Computer Aided Verification (CAV'11), 585-591, 2011.

[18] Z. Wang, S. Rose, and J. Huang, Securing DNS-Based CDN Request Routing, IEEE COMSOC MMTC Communications - Frontiers, Vol. 12, No. 2, 45-49, 2017.