The Hitchhiker's Guide to DNS Cache Poisoning

Sooel Son and Vitaly Shmatikov

The University of Texas at Austin

Abstract. DNS cache poisoning is a serious threat to today's Internet. We develop a formal model of the semantics of DNS caches, including the bailiwick rule and trust-level logic, and use it to systematically investigate different types of cache poisoning and to generate templates for attack payloads. We explain the impact of the attacks on DNS resolvers such as BIND, MaraDNS, and Unbound and their implications for several defenses against DNS cache poisoning.

Key words: DNS, cache poisoning, formal model

1 Introduction

The Domain Name System (DNS) is an essential part of the Internet. The primary purpose of DNS is to resolve symbolic domain names to IP addresses [10, 17, 18]. Many Internet security mechanisms, including host access control and defenses against spam and phishing, implicitly or explicitly depend on the integrity of the DNS infrastructure. Unfortunately, security was not one of the design considerations for DNS, and many attacks on DNS were reported over the years [3, 12, 15, 19].

Cache poisoning is arguably the most prominent and dangerous attack on DNS. DNS cache poisoning results in a DNS resolver storing (*i.e.*, caching) invalid or malicious mappings between symbolic names and IP addresses. Because the process of resolving a name depends on authoritative servers located elsewhere on the Internet (see Section 2.2), DNS protocol is intrinsically vulnerable to cache poisoning [3]. An attacker may poison the cache by compromising an authoritative DNS server or by forging a response to a recursive DNS query sent by a resolver to an authoritative server.

Many non-cryptographic defenses (surveyed in Section 8) focus solely on blind response forgery and attempt to solve the problem by increasing the entropy of DNS query components such as transaction IDs, query labels, and port numbers. This makes blind response forgery more difficult. Unfortunately, blind response forgery is just one of the possible attack vectors for DNS cache poisoning and, unlike cryptographic solutions, these defenses are vulnerable to trivial eavesdropping attacks. Therefore, they do not address the root causes of DNS cache poisoning and provide only partial protection.

Our goal is to develop a formal model for the semantics of DNS caches and use it to study cache poisoning attacks. Our analysis focuses on the *internal* operation of DNS resolvers and is thus complementary to the analyses of network protocols used to deliver DNS messages, success rate of forgery attempts, network-level defenses, and so forth. For instance, in a concurrent work [4], Bau and Mitchell formally modeled the cryptographic operations involved in the DNSSEC protocol, discovering a vulnerability that allows an attacker to add a forged name into a signed zone.

By contrast, we analyze the internal bailiwick rules (used by DNS resolvers to decide whether to accept a mapping from a given authority) and the trust levels of DNS data (used by resolvers to decide whether to overwrite an existing record). The bailiwick rule in particular, while critical for DNS security and reliability, is not part of the DNS standard and left to the resolver implementation. Its subtleties are often exploited by cache poisoning attacks, regardless of the actual mechanism (such as blind response forgery) used to deliver attack packets. To the best of our knowledge, internal operations of DNS resolvers have not been formally modeled before.

Our contributions. We explain the nature of DNS cache poisoning attacks and present a precise, formal model of the bailiwick rule and the record overwriting mechanism of modern DNS resolvers, including BIND v9.4.1, Unbound v1.3.4, and MaraDNS v1.3.07.09. We use our model to systematically enumerate and analyze different types of cache poisoning attacks and to explain the damage to different aspects of DNS resolution resulting from each attack. Using the ProVerif protocol analysis tool [6], we automatically construct attack templates for all attacks in our taxonomy, verifying that the attacks work against actual implementations.

The objective of this study is to develop a precise understanding of the semantics of modern DNS caches, including their bailiwick rules and trust-level logic. We do *not* propose a new defense against DNS response forgery since defending against specific types of DNS compromise is largely orthogonal to our goals. (In general, the only reliable protection is provided by cryptographic authentication schemes such as DNSSEC; unfortunately, these schemes are not yet deployed widely.) Instead, we use our model to enumerate the consequences of different types of DNS forgery exploits, regardless of whether they are perpetrated via server compromise, birthday attack, eavesdropping, or some other attack vector. We also show that our model is useful for evaluating the effectiveness of some non-cryptographic defenses against DNS response forgery.

2 DNS Background

2.1 Resource record set

DNS is a distributed storage system for Resource Records (RR). Each DNS resolver or authoritative server stores RRs in its cache or local zone file. A Resource Record includes a label, class, type, and data [10]. The label of an RR is a symbolic domain name used when accessing an Internet resource [17]. The class is either IN, or CH; the class of most RRs is IN, which means the Internet system. The type can have many possible values, but we will focus on records of type A, CNAME, and NS. An A record holds a mapping from a domain name to an IP address, a CNAME record holds a mapping from a domain name to an alias, and an NS record holds a mapping from a domain name to the name of an authoritative name server for that domain. Each record has a time-to-live (TTL) parameter and is purged from the cache once its TTL expires.

No two RRs in the cache may have the same label, class, type, and data, but it is possible to have multiple records with the same label, class, and type. Such a group is called a Resource Record Set (RRset).

2.2 Caching and recursive resolution

When a DNS resolver or authoritative server receives a query, it searches its cache for a matching label. If there is no matching label in the cache, the server may instead retrieve from the cache and return a referral response, containing an RRset of NS type whose label is "closer" to the domain which is the subject of the query [17].

Instead of sending a referral response, the DNS resolver may also be configured to initiate the same query to an authoritative DNS server responsible for the domain name which is the subject of the query [17]. Each query is identified by a random 16-bit transaction ID (TXID). The authoritative server can respond with an answer, a referral, or a failed response. In general, a response is comprised of the query, answer, authority, and additional sections. Each section may have none, one, or multiple RRsets.

The authoritative server's response—or a forged message pretending to be the authoritative server's response—is accepted by the DNS resolver and stored in its cache only if the RRset of each section passes a set of conditions known as the *bailiwick rule*. These conditions are not part of the DNS specification and depend on the implementation of the resolver. Furthermore, in certain circumstances (see Section 5), the received records may even *overwrite* those already stored in the cache.

Poisoning the DNS cache by adding false records is a serious threat, but DNS records corresponding to popular domains are likely to be already stored in the cache prior to an attack and are thus not vulnerable to the basic forgery exploit (this observation underlies the naive defense of increasing the time-to-live parameter of these records). It is the ability to overwrite existing records that makes DNS response forgery such a devastating attack. To understand record overwriting, we need to understand (1) the mechanism through which an attacker may introduce forged records into the cache of a DNS resolver (Section 3) and (2) the bailiwick and trust-level rules that govern addition and overwriting of records in DNS caches (Sections 4 and 5).

3 DNS Response Forgery

3.1 Cache poisoning without response forgery

Before BIND adopted the bailiwick rule in 1993, the owner of any DNS authoritative server could compromise records corresponding to any domain name [22, 23]. When responding to a query from the resolver, a malicious authoritative server can send, in the additional section of its response, an arbitrary mapping from any domain name (including those outside its authority) to an IP address.

For instance, consider a malicious authoritative server for `bad.com`. When a client asks its DNS resolver to resolve `www.bad.com`, the resolver queries the server. The server's response contains in its additional section the mapping from, say, `ns1.good.com` to a malicious IP address. Without the bailiwick rule (described in Section 4), this mapping would have been cached by the resolver, even though `good.com` was neither part of the query, nor under the malicious server's authority (see Fig. 2(a)).

3.2 Blind response forgery using birthday attack

The basic DNS protocol does not authenticate responses to recursive queries. The only checks are: (1) the query section and 16-bit transaction ID (TXID) of the response must match those of the query, and (2) the source IP address and destination port of the response must match, respectively, the destination IP address and source port of the query. The first arriving UDP packet which satisfies these conditions is treated as a valid response from the authoritative server.

Prior to recent patches [8], many DNS resolvers used a fixed port to send queries. Therefore, with the exception of a random TXID, all values used by the resolver to determine the validity of a packet received in response to its query are predictable. To generate a valid-looking response, it is sufficient to guess the TXID used in the query.

Attacks on DNS exploiting the "birthday paradox" have been known since at least 2002 [21]. If the TXID has only N bits of entropy (in practice, $N = 16$), a network attacker needs only $O(2^{\frac{N}{2}})$ trials on average to generate a forged response which matches the TXID of the query and will thus be accepted as valid by the target resolver. The answer section of the forgery contains a malicious mapping from a domain name to an IP address (see Fig. 1).

For the attack to succeed, the forgery must arrive to the target resolver before the response from the legitimate authoritative server. If the legitimate response arrives first, it will be cached by the resolver and until its time-to-live (TTL) expires, the resolver will not ask the authoritative server to resolve the same domain name, preventing the attacker from poisoning the mapping for that domain.

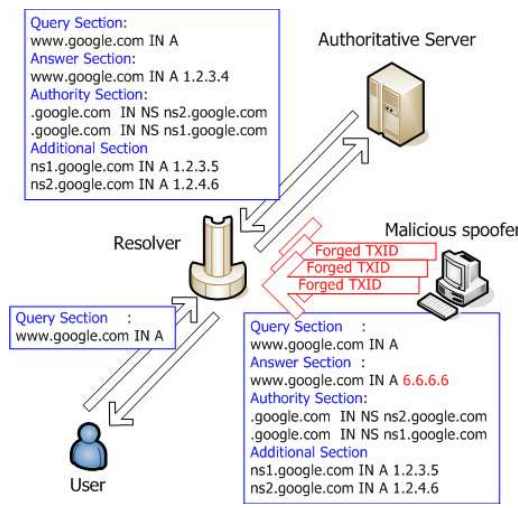

Fig. 1. Overview of the cache poisoning attack

```
Query Section    : www.bad.com IN A
Answer Section  : www.bad.com IN A 6.6.6.6
Authority Section:
.bad.com  IN NS ns1.yahoo.com
.bad.com  IN NS ns1.google.com
Additional Section
ns1.yahoo.com IN A 6.6.6.6
ns1.google.com IN A 6.6.6.6
```

(a) A forged response without considering bailiwick

```
Query Section    : xyx12.google.com IN A
Answer Section  :
NONE
Authority Section:
.google.com  IN NS www.google.com
Additional Section
www.google.com IN A 6.6.6.6
```

(b) Kaminsky's exploit

Fig. 2. Payloads of various cache poisoning attacks

Kaminsky's exploit. At Black Hat 2008, Kaminsky presented a new extension of the birthday attack [13]. While the basic mechanism is the same (using the birthday attack to forge a response with the same transaction ID as the query), three observations make Kaminsky's attack more serious than "conventional" DNS forgery [19].

First, the attacker can force the target resolver to initiate a query to an authoritative server of his choice. Second, modern attackers have enough network bandwidth to generate a large number of spoofed responses, each with a different guess of the transaction ID. Third, the malicious "payload" of the forged response is the additional section (as opposed to the answer section in the conventional attack), for reasons explained below.

The basic scheme of the exploit is as follows. The attacker chooses the domain name that he wants to compromise (*e.g.*, `www.google.com`). He then queries the target resolver with any subdomain which is not already cached on the resolver (*e.g.*, a non-existent subdomain such as `xyz12.google.com`). Because the name is not in the cache, this causes the target resolver to send a query to the authoritative server(s) for this domain. At this point, the attacker floods the resolver with a large number of forged responses, each containing a different guess of the query's transaction ID.

A typical forged response is shown in Fig. 2(b). Note that it is a *referral*, not an answer, and the false information is contained in the additional section rather than the answer section. This greatly increases the efficacy of the attack. Instead of hijacking a mapping for a single domain name, the attack, if successful, introduces into the target resolver's cache a false mapping for an authoritative server. Future queries from the compromised resolver will be sent directly to an attacker-controlled IP address, enabling the attacker to provide malicious responses without blind response forgery.

If a forgery attempt fails, the attacker can immediately start a new race, using a different domain name, and continue until he actually wins the race, *i.e.*, a forgery with a valid transaction ID arrives to the resolver before the legitimate answer.

3.3 Response forgery using eavesdropping

A number of recently proposed defenses against DNS cache poisoning, including source port randomization, 0x20-bit encoding, XQID, and WSEC-DNS, fundamentally depend on the *asymmetric accessibility* of the components used for authenticating responses to DNS queries [7, 8, 11, 20].

These defenses ensure neither confidentiality of DNS queries, nor authentication of responses (in contrast to cryptographic defenses such as DNSSEC) and thus prevent only *blind* forgery. DNS remains vulnerable to trivial attacks by compromised servers and/or network eavesdroppers: in a non-switched subnet environment, the attacker can run an eavesdropping tool in the promiscuous mode; in a switched environment, ARP poisoning [14] or any similar technique can be used to force all packets from the target resolver to pass through a malicious computer on the same subnet.

4 The Bailiwick Rule

The purpose of the bailiwick rule is to prevent malicious authoritative servers from providing DNS mappings for domains outside their authority as part of a referral response (see Section 3.1). For example, the authoritative server for `.com` can return a mapping for any `.com` domain, while the authoritative server for `bad.com` should only be able to provide mappings for subdomains of `bad.com`.

The RFC specifying the DNS protocol does not define a concrete bailiwick rule. For the purposes of this paper, we analyze the bailiwick rules of three open-source implementations: BIND v9.4.1 [1], Unbound v1.3.4 [16], and MaraDNS v1.3.07.09 [24].

BIND. The complete bailiwick-checking algorithm of BIND v9.4.1 is shown in Fig. 5 in the appendix. The key data structure used to keep track of the bailiwick at any given point in the recursive DNS resolution is `Query.zone`. If a BIND resolver cannot resolve a query locally, it finds the RRset whose label is the "closest" to the received query among all RRsets of type NS in its cache. This label is stored in `Query.zone`, and the resolver sends the query to the name server indicated by the NS record.

If the response holds an RRset in the answer section, the resolver caches it after checking that its domain label matches the query. NS records in the authority section are cached only if their domain label is a subdomain of or equal to `Query.zone`.

If the response is a referral (*i.e.*, the answer section does not contain a record), the resolver must resend the query to another name server, as indicated in the referral. At this point, the bailiwick check is performed. First, the resolver checks whether the domain label of the query is a subdomain of the label in the authority section of the received response. If it is, the resolver next checks whether the domain label in the authority section is a subdomain of the current value of `Query.zone`. Only if both conditions hold, the resolver caches the NS-type RRset received in the referral.

The next step is to determine whether to cache the RRsets in the additional section of the referral. If the domain label of each record in the additional section is a subdomain of `Query.zone`, the additional section is cached; otherwise, it is not cached and the resolver will initiate new queries for the labels of the records from the additional section.

This prevents a malicious name server from referring a query to a name server in a different bailiwick along with a false IP address mapping for that server (as in Fig. 2(a)), since the resolver will not cache the additional section containing the false mapping.

Finally, the value of `Query.zone` is changed to the label of the RRset in the authority section of the received referral response. The BIND resolver then initiates queries for the names which were not cached in the previous steps.

Unbound. The bailiwick checking algorithm of Unbound (Fig. 6 in the appendix) is very different from BIND. All records whose labels are out of the bailiwick are removed from the received responses. The remaining records are cached, provided the response contains at least one answer record which comes from an authoritative server.

Unbound also differs in how it decides whether a response is a referral or not. If the label of the authority record in a response is below the resolver's bailiwick zone, Unbound labels the response as a referral even if there is a record in the answer section. The resolver caches all records from the additional and authority sections of a referral response, but, by default, does not send them to clients [26].

MaraDNS. The bailiwick logic of MaraDNS (Fig. 6 in the appendix) is significantly simpler. MaraDNS does not cache the authority and additional section of responses containing an RRset in the answer sections, thus eliminating the need to perform bailiwick checks on them. Furthermore, even for referral responses, MaraDNS caches neither the NS mapping from the domain name to an authoritative server name (authority section), nor the A mapping from the latter name to an IP address (additional section). Instead, MaraDNS simply stores an authority section with a mapping from the domain name to the IP address. This eliminates the need to perform a bailiwick check on the name of the authoritative server, since this name is not cached (with a potential loss in efficiency).

Differences between resolver implementations. Table 1 summarizes the differences between DNS resolver implementations. There is an important difference between BIND's and Unbound's default caching policies for RRsets in the additional section. To shorten query resolution times, BIND caches all such mappings, including domain labels and IP addresses from the additional section. By default, Unbound, too, caches the additional section, but these mappings are not sent to the client as the answer for a query. Therefore, from the client's viewpoint, the default behavior of Unbound is similar to that of MaraDNS. All three implementations can be compromised by different types of cache poisoning attacks (the implementations are semantically correct, but the protocols they use for updating the DNS cache are intrinsically weak). In Section 7, we show which implementation is vulnerable to which attack.

Functionality	BIND 9.4.1	Unbound 1.3.4	MaraDNS 1.3.07
RRset trust rules	O	O	X
Caching answer section	O	O	O
Caching authority section from a referral response	O	O	O
Caching authority section from an answer response	O	O	X
Caching additional section from a referral response	O	O	X
Caching additional section from an answer response	O	O	X
Additional section data sent to clients	Default O	Default X	X

Table 1. Differences between resolver implementations.

Kaminsky's exploit and the bailiwick check. Kaminsky's exploit does not violate the bailiwick rule. The forged referral in this attack contains an authority section with a (possibly fake) *in-bailiwick* name server, along with an additional section mapping this server to an attacker-controlled IP address. This invalid mapping is cached by the target BIND resolver. If the attacker wants to compromise the mapping of an existing name server (as opposed to introducing a fake one), there is a complication. The mappings for the name servers of popular domains tend to have long TTLs; they are likely to be already present in the victim's cache and must be overwritten. In Section 5, we explain the conditions under which an existing record may be overwritten.

Unbound caches RRsets from the additional section, but, by default, does not send them to clients. These RRsets are used internally by the resolver to find IP addresses of authoritative servers and can be overwritten, facilitating certain attacks (see Section 7).

MaraDNS will accept the malicious authority section, but the mapping from the fake name server to an attacker-controlled IP address will not be cached. The IP address of an authoritative server can be changed only by overwriting an existing mapping.

5 Cache Overwriting

Cache poisoning attacks are especially dangerous because they enable the attacker not just to add false mappings to the cache of vulnerable DNS resolvers, but also to *overwrite* existing mappings, including long-lived mappings for popular domains.

The rules for overwriting cache records are defined in RFC 2181 [10]. They depend on the *trust level* of an RRset. Table 2 shows trust levels used by BIND resolvers. The trust level of an RRset contained in a response depends on whether it comes from an authoritative server and whether the response is a referral. Trust level 8 is for records in a local zone setup file provided by a DNS server administrator, while trust level 7 is used by DNSSEC. We focus on records whose trust levels are from 2 to 6.

Define symbol	Trust level	Description
dns_trust_ultimate	8	This server is authoritative
dns_trust_secure	7	Successfully DNSSEC validated
dns_trust_authanswer	6	Answer from an authoritative server
dns_trust_authauthority	5	Received in the auth section as an authority response
dns_trust_answer	4	Answer from a non-authoritative server
dns_trust_glue	3	Received in a referral response
dns_trust_additional	2	Received in the add section of a response

Table 2. Trust levels in BIND 9.4.1.

BIND and Unbound. In BIND, a cached RRset is overwritten if the trust level of the received RRset is higher or equal to the cached one and its TTL is longer. NS-type RRsets received in a referral are an exception: they have the trust level 8 for the purposes of overwriting (*i.e.*, they always overwrite the records already present in the cache), but are stored with the trust level 3.

In Unbound, the absolute trust levels are different, but the relative order is the same. Therefore, we use the same trust-level model for BIND and Unbound.

MaraDNS. MaraDNS does not use trust levels. A new record contained in the response simply overwrites the existing record. In practice, however, only NS records can be overwritten by forged responses. Because MaraDNS does not cache the additional section of responses, in order to overwrite an A or CNAME record the forged response should contain the replacement mapping in the answer section and its label must be exactly the same as the label of the record to be overwritten. Such a forgery would only be accepted in response to a query with the same label. Observe, however, that since a record with this label is already present in the cache, a MaraDNS resolver would never initiate a recursive query for this label. Therefore, there is no query that would give the attacker an opportunity to overwrite an existing A or CNAME record.

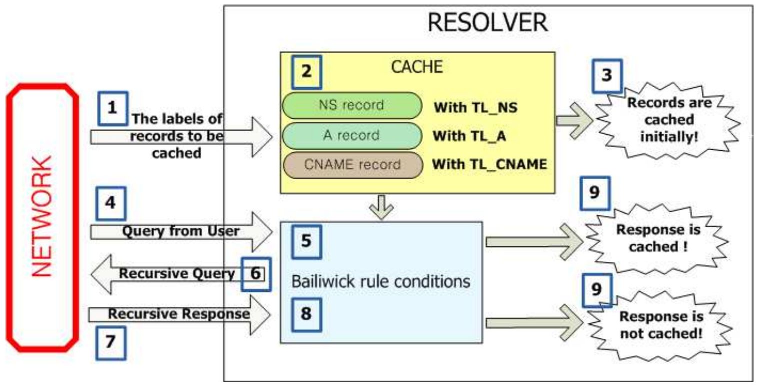

Fig. 3. Generic model of a DNS resolver

6 Formal Model of DNS Resolver

6.1 Modeling methodology

As shown in Sections 4 and 5, the semantics of DNS caches are quite complicated. To understand the potential impact of cache poisoning attacks, we construct a formal model of the default bailiwick-checking and cache-overwriting rules of BIND v9.4.1 and Unbound v1.3.4. We do not build a formal model for MaraDNS because it does not cache the authority and additional sections of responses containing an answer RRset and does not use trust levels for overwriting existing records. We do show attacks against all three implementations, including MaraDNS, in Table 4

We use the ProVerif protocol analysis tool, due to its success in practical formal verification of security protocols (*e.g.*, [2]). The details of ProVerif are beyond the scope of this paper and can be found in [6]. The behavior of each protocol participant is modeled as a set of Horn clauses, which represent sending or receiving messages on specified communication channels. ProVerif then uses a sound, resolution-based algorithm to determine whether a specified property holds over all executions of the protocol.

Fig. 3 shows the abstract model of DNS resolver. We use it to check whether or not a cached resource record with a certain label and trust level is secure against cache poisoning conducted by the active adversary who has complete control of the network. This attacker model may appear strong, but we emphasize once again that our goal is to model the *internal behavior of DNS resolvers*, not the details of the network protocol through which DNS messages are exchanged. By modeling the network as a public channel, we can focus on the semantics of the cache and abstract from the particulars of the forgery method through which attacker packets are introduced.

The initial state of the model asserts that three valid records of types A, NS, and CNAME, respectively, have been cached. Their labels are determined via network inputs. (In reality, the attacker can insert an arbitrary label into the cache by tricking a client of the resolver into asking to resolve the corresponding name.) Trust levels are specified manually. The model then receives a query from the network. If recursive resolution is required, the model sends out a recursive query and receives a response from the network. The bailiwick rule in the model determines which records in the response should be cached. If a malicious record satisfies the bailiwick conditions, the model asserts that a cache poisoning event has occurred.

6.2 Base data types

We use a simplified model of DNS records with only three components—type (A, NS, or CNAME), domain name, and data—and ignore other aspects such as authority RRsets in the answer section, lame resolution, and zone delegation. Events model critical points in the resolution process. The *evInitCache* event occurs when the model is initialized. It asserts that a record is cached with a verifier-specified trust level prior to the attack (in our analysis, we vary the trust level to determine whether or not a particular record can be overwritten). The *evRecursiveQueryStart* event occurs when the cache does not have a record matching a given query and the resolver must send a recursive query to an authoritative server responsible for the bailiwick zone. The *evPoison* event occurs when an invalid record passes all checks and is about to be cached.

6.3 Cache initialization

Our model assumes that the CNAME, A and NS resource records for a certain name are already present in the cache. The model then generates a query, waits for a response from the network, and decides whether or not the response should be cached.

The following property says that the *evPoison* event does not occur unless the *evInitCache* event has occurred. More precisely, in the resolver which already caches *cachedns*, *cacheda* and *cachedcname* labels, a resource record whose label is *poisonedlabel* and whose type is *rectype* is cached with the trust level *tl* only if there has occurred an *evInitCache* event in which the resource record whose label was *cachedlabel* and whose type was *cachedtype* was cached with the trust level *cachedtl*.

$$\begin{aligned}&\text{query ev: evPoison(rectype, poisonedlabel, poisoneddata, tl,}\\&\qquad\qquad\qquad\text{cachedns, cacheda, cachedcname)}\\&\longrightarrow \text{ev: evInitCache(Record(cachedtype, cachedlabel, data), cachedtl)}\end{aligned}$$

6.4 Non-overwritability

Recall from Section 5 that a cached record can only be overwritten by a record with an equal or higher trust level. The following properties model "non-overwritability" of A records with various trust levels:

$$\begin{aligned}&\text{query ev: evPoison(At, cacheda, wrongdst, tl, cachedns, cacheda, cachedcname)}\\&\quad\longrightarrow \text{ev: evInitCache(Record(At, cacheda, validdst), tl6)} \wedge \text{tl6} > \text{tl}\\&\text{query ev: evPoison(At, cacheda, wrongdst, tl, cachedns, cacheda, cachedcname)}\\&\quad\longrightarrow \text{ev: evInitCache(Record(At, cacheda, validdst), tl4)} \wedge \text{tl4} > \text{tl}\\&\text{query ev: evPoison(At, cacheda, wrongdst, tl, cachedns, cacheda, cachedcname)}\\&\quad\longrightarrow \text{ev: evInitCache(Record(At, cacheda, validdst), tl3)} \wedge \text{tl3} > \text{tl}\\&\text{query ev: evPoison(At, cacheda, wrongdst, tl, cachedns, cacheda, cachedcname)}\\&\quad\longrightarrow \text{ev: evInitCache(Record(At, cacheda, validdst), tl2)} \wedge \text{tl2} > \text{tl}\end{aligned}$$

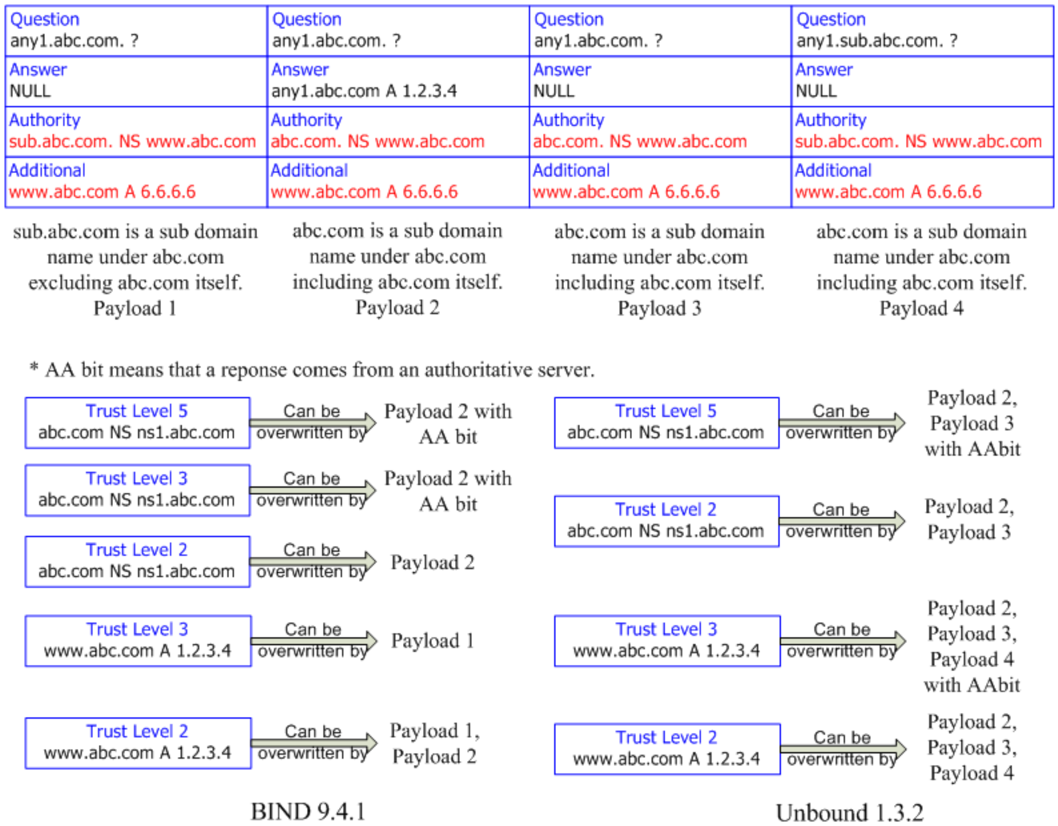

Fig. 4. All ways to overwrite an existing RRset in the cache.

Each of these properties says that whenever a poisoning event occurs, the target record is already cached with a certain trust level which is higher than the trust level of the forged response. If the property holds, the existing record cannot be overwritten. If the property cannot be provable, then the model contains at least one path in which the trust level of the forged record is higher than the trust level of the cached record. Therefore, the cached record can be successfully overwritten by the forgery.

ProVerif analysis shows that in both BIND and Unbound, non-overwritability holds only for trust levels 4 and 6. All cached records whose trust level is 2, or 3 *can* be overwritten. For all interesting trust levels of an A or NS record, Fig. 4 shows the (automatically generated) templates for malicious payloads to be used in the forged response. In Fig. 4, we assume that the NS record of `abc.com` and the A record of `www.abc.com` are already cached by the victim resolver.

The following property is always false in our model, showing that CNAME records cannot be overwritten.

query ev: evPoison(CNAMEt, cachedc, invalid, tl, cachedns, cacheda, cachedc)

6.5 Bailiwick rule

The primary purpose of the bailiwick rule is to prevent an authoritative server from claiming the mappings from domain names belonging to other authorities. To determine whether the bailiwick-checking logic of BIND and Unbound resolvers achieves this, we used ProVerif to verify the following three properties:

query ev: evPoison(NSt/At/CNAMEt, targetname, dst, tl, cachedns, cacheda, cachedc)
$\longrightarrow$ ev: evRecursiveQueryStart(query, bailiwick, bailiwickAAserver)
$\wedge$ isSubName: query, bailiwick
$\wedge$ isSubName: targetname, bailiwick

These properties say that a record can enter the cache (represented by the cache poisoning event, since in our model all responses arrive from the network attacker) only in response to a recursive query and if *targetname* and *query* are subdomains of *bailiwick*. Here *bailiwick* is the authority name closest to the domain label in the query.

According to ProVerif, these three properties hold in our model. Therefore, the domain name of both legitimate and forged responses must be a subdomain of the proper bailiwick, as determined by the DNS resolver. Note, however, that the bailiwick depends on the label of the current query. An attacker may initiate a query for a domain of his choice or manipulate the resolver into issuing such a query (*e.g.*, by tricking one of the resolver's users into visiting a webpage with a link to the domain), thus ensuring that forged responses do not violate the bailiwick rule.

7 Taxonomy of Cache Poisoning Attacks

We use our model to systematically enumerate several types of cache poisoning attacks, the corresponding payloads of forged DNS responses, and their effect on the compromised resolver. Our taxonomy is shown in Table 3. It is *complete* for A, NS, and CNAME records. We assume that the resolver has already fixed the bailiwick zone (abc.com) for incoming responses. Every name which can be a target of cache poisoning belongs to one of three categories: domain outside abc.com, subdomain of abc.com, or abc.com itself. There are two types of cache poisoning: adding a new name and overwriting the mapping for an existing name. Table 3 covers all possibilities.

Table 4 summarizes the feasibility of different types of cache poisoning attacks against different resolver implementations. Because BIND and Unbound use different caching policies by default and MaraDNS does not cache the additional section, the effective attack payload varies from resolver to resolver. For BIND and Unbound, our analysis is based on our formal model and experimental attacks against the resolver implementation. For MaraDNS, we analyzed the bailiwick-checking logic manually (it is significantly simpler than in either BIND, or Unbound).

7.1 Adding a new CNAME record

Our model shows that the only way to add a malicious CNAME mapping to the cache is to forge an answer section whose label is exactly same as the query (the reason is

Target domain name	Type	Type of poisoning	
		Adding a new mapping	Overwriting an existing mapping
Domain name outside abc.com	CNAME A NS	Impossible (Section 6.5)	Impossible (Section 6.5)
abc.com	NS	Target name is already in the cache	Possible (Section 7.4)
Subdomain of abc.com	CNAME	Possible (Section 7.1)	Impossible (Section 6.4)
	A	Possible (Sections 7.1, 7.2, 7.5, 7.6)	Possible for trust levels 2, 3 (Sections 7.3, 6.4)
	NS	Possible (Section 7.4)	Possible (Section 7.4)

Table 3. Taxonomy of cache poisoning attacks on BIND and Unbound (abc.com is the bailiwick zone).

Type of attack	BIND 9.4.1	Unbound 1.3.4	MaraDNS 1.3.07
Adding a new CNAME record (Section 7.1)	Effective	Effective	Effective
Adding a subdomain under an existing authority (Section 7.2)	Effective	Possible, but ineffective with the default policy	Impossible by forging additional data
Overwriting an existing A record (Section 7.3)	Effective	Effective	Impossible *
Overwriting an existing NS record (Section 7.4)	Effective	Effective	Effective
Creating fake domains (Section 7.5)	Effective (by forging additional section)	Effective (requires prior overwriting of IP addresses of authoritative servers)	Effective (requires prior overwriting of IP addresses of authoritative servers)
Stealing a popular domain name by hijacking subauthorities (Section 7.6)	Effective	Effective	Effective

* IP addresses of authoritative servers can be overwritten without overwriting an A record.

Table 4. Cache poisoning attacks on different resolvers. All attacks have been tested against actual implementations.

that the authority section contains only NS records and the additional section only A records). This is captured by the following property:

$$\text{query ev: evPoison(CNAMEt, newname, invalidlabel, tl, cachedns, cacheda, cachedc)} \\ \longrightarrow \text{ev: evInitCache(Record(At, cacheda, validlabel), cachedtl)}$$

The disadvantage of this attack is that it cannot be easily perpetrated via blind, brute-force forgery. If the attacker fails in a single race, the resolver will cache the failed label and the attacker must change the target name. If, however, the attacker poisons the IP addresses of authoritative servers for a certain zone, he controls all names in this zone and adding any CNAME mapping is trivial. The IP addresses of authoritative servers are usually cached with the trust level 2 or 3 and can thus be overwritten (Section 7.3).

7.2 Adding a subdomain under an existing authority

This exploit adds a record for a fake subdomain under an existing authority in the victim's cache. It is modeled by the following property:

query ev: evPoison(At, makeSubName(bad, goodZone), invalid, tl,
goodZone, makeSubName(good, goodZone), cname)
$\longrightarrow$ ev: evInitCache(Record(At, makeSubName(good, goodZone), valid), cachetl)

As shown in Fig. 4, payloads 1 and 2 can add a new domain name to a BIND cache. By default, the RRsets in the additional section will be used as the answer to the query. Payloads 2, 3, and 4 can add a new domain name to an Unbound cache, but Unbound's default policy does not send this information to clients.

This attack is dangerous to clients using BIND resolvers because many Web security policies are vulnerable to attacks from subdomains. For example, many websites set the path and domain name of cookies as, respectively, '/' and the top two levels of the site's domain (*e.g.*, `example.com` rather than `www10.example.com`). An attacker who uses cache poisoning to introduce a fake subdomain can use phishing to lure naive users to this subdomain and then overwrite and/or read cookies set by legitimate subdomains.

7.3 Overwriting an existing A record

One may assume that address mappings for popular domain names are already cached by most resolvers with the trust level 4 or 6. Therefore, they cannot be overwritten until their TTL expires. This is the basis of a common defense against DNS forgery: simply increase TTL for legitimate DNS records.

A cleverer attack exploits the fact that it is uncommon for clients to directly initiate queries about authoritative name servers such as `ns1.google.com`. Records with addresses of authoritative name servers are typically received by resolvers as part of referral responses, which are cached with the trust level 2 or 3. Therefore, they can be overwritten. In our model, this is captured by the following property:

query ev: evPoison(At, targetname, invaliddata, tl, ns, targetname, cname)
$\longrightarrow$ ev: evInitCache(Record(At, targetname, validdata), tl2) $\wedge$ tl2 $>$ tl
query ev: evPoison(At, targetname, invaliddata, tl, ns, targetname, cname)
$\longrightarrow$ ev: evInitCache(Record(At, targetname, validdata), tl3) $\wedge$ tl3 $>$ tl

Our formal analysis shows that payloads 1 and 2 for BIND and payloads 2, 3, and 4 for Unbound (see Fig. 4) can accomplish this attack.

This attack is dangerous to clients of both BIND and Unbound. It results in changing the IP addresses of authoritative servers and enables the attacker to compromise *any* domain in the server's zone. Furthermore, IP address mappings for the names of root DNS servers such as `A.ROOT-SERVERS.NET` can be stored in the cache with the trust level 2. Although there are only 13 root servers, making forgery harder, if the attack does succeed, their addresses can be overwritten.

7.4 Overwriting an existing NS record

Unlike Kaminsky's attack, which uses the authority and additional sections of the forged response to compromise the mapping from a domain name to an IP address, forged responses can also be used to overwrite existing NS records in the resolver's cache [25]. In our formal model, this is represented by the following property:

query ev: evPoison(NSt, targetname, invalidlabel, tl, targetname, a, cname)
$\longrightarrow$ ev: evInitCache(Record(NSt, targetname, validlabel), cachetl) $\wedge$ cachetl $>$ tl

Payload 2 from Fig. 4 works against BIND, payloads 2 and 3 against Unbound.

The consequence of this attack is that any query for a domain name under the compromised authority is sent by the resolver directly to an attacker-controlled authoritative server(s). This exploit is more serious than Kaminsky's exploit because it effectively hijacks *every* domain name under the compromised authority. We emphasize that the attacker can overwrite any NS record in the cache, even those with non-expired TTL.

7.5 Creating fake domains

Cache poisoning enables the attacker to insert a mapping for any domain name into the victim resolver's cache even if the domain does not exist in reality. For example, the attacker can create mappings for plausible domain names such as `www.google.edu` and `www.university.gov`, making it easier to carry out phishing attacks. To stage this exploit, the forged packet must look like a valid response from the authoritative server for a top-level domain such as `.edu` or `.gov`. Against BIND, it is sufficient to forge RRsets in the additional section. Technically, the attack is modeled by the same rules and uses the same payloads as in Section 7.2.

The attack against Unbound is more sophisticated because Unbound by default does not send the additional section to clients. The attacker must change the authority section for the target zone or the IP addresses of the zone's authoritative servers. Once that's done, adding a new name under this zone is trivial. Technically, this attack is modeled by the same rules and uses the same payloads as in Section 7.3 (respectively, 7.4).

7.6 Hijacking a popular domain via a sub-authority

A common objective of DNS attacks is to compromise the mappings for popular domain names such as `www.paypal.com` and `www.google.com`. As mentioned above, such mappings are difficult to compromise because they are likely to be already cached with a long TTL. In practice, popular domain names are usually mapped to subdomains via long-lived CNAME records. For example, `www.google.com` may be mapped to `www.l.google.com`. Even if the attacker succeeds in forging an A record which maps `www.google.com` to a malicious IP address, the resolver will use the unexpired CNAME record rather than the forged A record, foiling the attack.

Subdomain names, however, are mapped to actual IP addresses by A records with relatively short TTL values. For example, the record mapping `www.l.google.com` to an IP address may have a 300-second TTL. Suppose the attacker poisons the authority section for `l.google.com`. Once the A record for `www.l.google.com` expires, the victim will ask an attacker-controlled server to resolve `www.l.google.com`, giving him complete control over the mapping. This attack is effective against both BIND and Unbound because it targets the authority section of a zone or the IP address of the zone's authoritative server, not the records in the additional section. Therefore, Unbound's default policy does not prevent the attack. Technically, this attack is modeled by the same rules and uses the same payloads as in Section 7.3 (respectively, 7.4).

8 Defenses

The objective of our formal model is to understand the nature and impact of cache poisoning attacks at the level of DNS resolvers, *not* the protocol through which poisoned packets are delivered. By contrast, the defenses surveyed below (with the exception of cryptographic defenses) focus solely on preventing blind response forgery, which is simply one of the many vectors for cache poisoning attacks. Therefore, they are largely complementary and orthogonal to the goals of this paper.

Cryptographic solutions include DNSSEC [9] and DNSCurve [5]. DNSSEC uses digital signatures to authenticate and protect integrity of responses to DNS queries. So far, cryptographic solutions have not been widely deployed due to their impact on DNS performance, as well as political and infrastructural issues.

The most popular non-cryptographic defense against blind response forgery is UDP source port randomization [8]. It increases entropy of recursive DNS queries by randomizing the source port number in addition to the transaction ID, thus making the birthday attack more difficult. This patch depends on the configuration of the local network such as the firewall imposing strict constraints on inbound connections. Other solutions aiming to prevent blind response forgery by increasing entropy of queries are 0x20-bit encoding [7], which randomizes capitalization of letters in the query (the amount of entropy depends on the length of the query), and WSEC-DNS [20] and XQID [11], which use a challenge-response scheme with random nonces.

While these solutions may be effective for blocking a particularly dangerous attack vector (namely, blind response forgery), they do not actually authenticate responses to recursive DNS queries and should be viewed only as a temporary patch until proper authentication mechanisms are deployed. As long as there exist other attack vectors (see Section 3) and modern resolver implementations such as BIND and Unbound cache information provided in the authority and additional sections of unauthenticated responses (see Section 4), DNS cache poisoning will remain a serious issue.

Other proposed solutions include increasing TTLs of legitimate records and limiting the number of simultaneous recursive queries (the latter to decrease the number of simultaneous races that may be staged by the attacker). Our model helps evaluate such defenses because their efficacy depends on a detailed understanding of the semantics of DNS caches. For example, our analysis shows that increasing TTL does not help against a large class of attacks that involve overwriting of existing DNS records.

9 Conclusion

We presented a formal model of DNS cache semantics, including the bailiwick and trust-level rules used by common resolver implementations, and analyzed it with the ProVerif protocol analysis tool. The result is a comprehensive taxonomy of cache poisoning attacks, showing (1) which parts of the cache can be poisoned, (2) conditions necessary for each attack, and (3) consequences of each attack. Furthermore, our analysis enabled us to produce payload templates for each attack. We argue that our formal model is an essential tool for understanding the subtle caching rules used by modern DNS resolvers and developing robust defenses against DNS cache poisoning.

References

1. Internet Systems Consortium BIND 9.4.1. `http://www.isc.org/downloadtables`.
2. M. Abadi and B. Blanchet. Computer-assisted verification of a protocol for certified email. *Sci. Comput. Program.*, 58(1-2):3–27, 2005.
3. D. Atkins and R. Austein. Threat Analysis of the Domain Name System (DNS). RFC 3833 (Informational), August 2004.
4. J. Bau and J. Mitchell. A security evaluation of DNSSEC with NSEC3. In *NDSS*, 2010.
5. D.J. Bernstein. DNSCurve. `http://DNSCurve.org`.
6. B. Blanchet. Automatic verification of correspondences for security protocols. *J. Computer Security*, 2009.
7. D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee. Increased DNS forgery resistance through 0x20-bit encoding. In *CCS*, 2008.
8. C. R. Doughety. Vulnerability note vu#800113, 2008. `https://www.kb.cert.org/vuls/id/800113`.
9. D. Eastlake. Domain Name System Security Extensions. RFC 2535 (Proposed Standard), March 1999. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 2931, 3007, 3008, 3090, 3226, 3445, 3597, 3655, 3658, 3755, 3757, 3845.
10. R. Elz and R. Bush. Clarifications to the DNS Specification. RFC 2181 (Proposed Standard), July 1997. Updated by RFCs 4035, 2535, 4343, 4033, 4034.
11. J. Høy. Anti DNS spoofing - extended query ID (XQID), April 2008. `http://www.jhsoft.com/dns-xqid.htm`.
12. C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from DNS rebinding attacks. In *CCS*, 2007.
13. D. Kaminsky. Black ops 2008-it's the end of the cache as we know it. Presented at BlackHat 2008, 2008.
14. T. King. Packet sniffing in a switched environment, August 2002. `http://www.sans.org/reading_room/whitepapers/networkdevs/`.
15. A. Klein. BIND 9 DNS cache poisoning, March 2007. `http://www.trusteer.com/bind9dns`.
16. NLnet Labs. Unbound 1.3.4. `http://www.unbound.net/download.html`.
17. P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592.
18. P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343.

19. M. Olnet, P. Mullen, and K. Miklavcic. Dan Kaminsky's 2008 DNS vulnerability, 2008. http://www.ietf.org/mail-archive/web/dnsop/current/pdf2jgx6rzxN4.pdf.
20. R. Perdisci, M. Antonakakis, X. Luo, and W. Lee. WSEC DNS: Protecting recursive DNS resolvers from poisoning attacks. In *DSN-DCCS*, 2009.
21. V. Sacramento. Vulnerability in the sending requests control of Bind version 4 and 8 allows DNS spoofing, November 2002. http://www.rnp.br/cais/alertas/2002/cais-ALR-19112002a.html.
22. C. Schuba. Addessing weaknesses in the domain name system protocol, 1993. http://ftp.cerias.purdue.edu/pub/papers/christoph-schuba/.
23. Secure Works. DNS cache poisoning - the next generation, 2007. http://www.secureworks.com/research/articles/dns-cache-poisoning.
24. S. Trenholme. MaraDNS 1.3.07.09. http://www.maradns.org.
25. Computer Academic Underground. http://www.caughq.org/exploits/CAU-EX-2008-0003.txt.
26. W. Wijngaards. Resolver side mitigations, August 2008. http://tools.ietf.org/html/draft-wijngaards-dnsext-resolver-side-mitiga%tion-00.

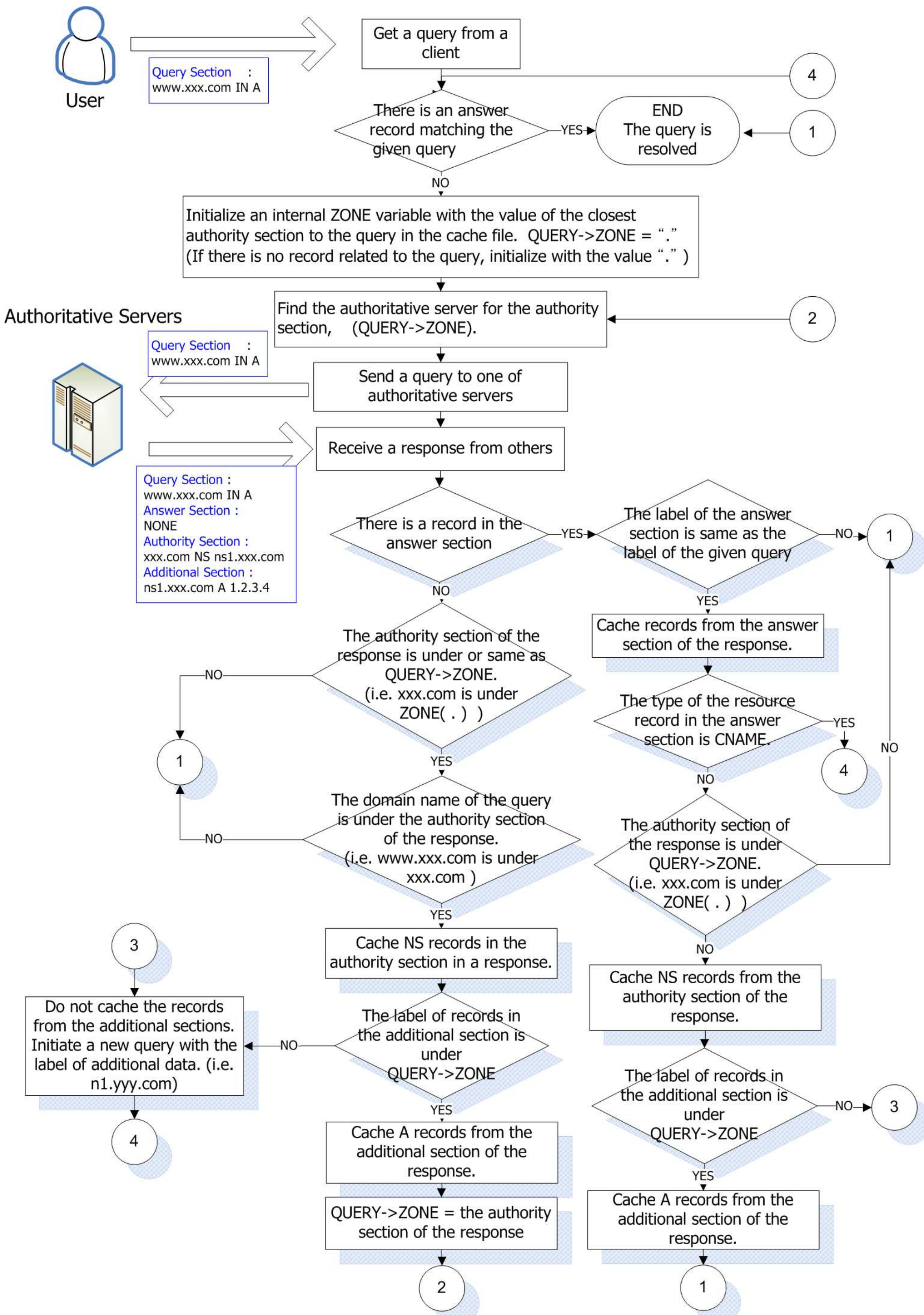

Fig. 5. Bailiwick-checking algorithm of BIND (shaded shapes are implementation-specific).

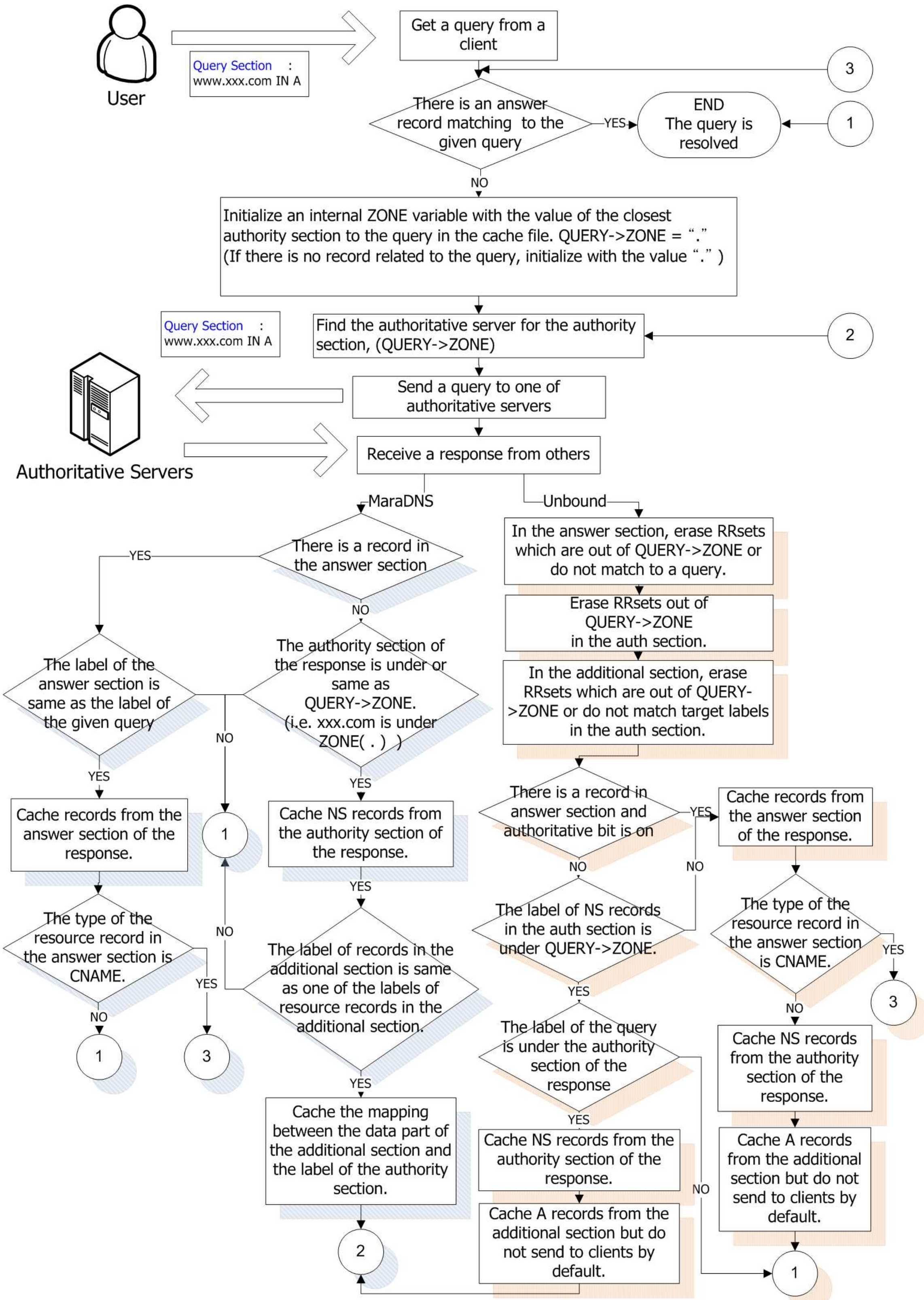

Fig. 6. Bailiwick-checking algorithm of Unbound and MaraDNS (shaded shapes are implementation-specific).

A BIND resolver model

```
(************************************************************
 *                                                          *
 *       BIND DNS resolver verifier                         *
 *                                                          *
 ************************************************************)

param redundantHypElim = beginOnly.
param traceBacktracking = false.
param traceDisplay = long.

free net.
fun NSt/0. fun At/0. fun CNAMEt/0.
fun zeroStage/0. fun oneStage/0.

data ValidIP/1. data InvalidIP/1.
data ValidDN/1. data InvalidDN/1.
data AA/1. data nil/0. data Record/3.

reduc GetSrc    ( Record( dmnSrc, target, recType ) ) = dmnSrc.
reduc GetDst    ( Record( dmnSrc, target, recType ) ) = target.
reduc GetType    ( Record( dmnSrc, target, recType ) ) = recType.
reduc GetDnDst  ( Record( dmnSrc, ValidDN(target), recType ) ) = target.

data Response/4.
reduc GetAnswer   ( Response( q1, Record(a1,a2,a3), Record(au1,au2,NSt), Record(ad1,ad2,At) ) )
= Record(a1,a2,a3).
reduc GetAuth     ( Response( q1, Record(a1,a2,a3), Record(au1,au2,NSt), Record(ad1,ad2,At) ) )
= Record(au1,au2,NSt).
reduc GetAdd      ( Response( q1, Record(a1,a2,a3), Record(au1,au2,Nst), Record(ad1,ad2,At) ) )
= Record(ad1,ad2,At).
data emptyset/0.

(* Predicate checking whether it is valid or not *)
pred checkrec/1.
clauses
  checkrec: Record( x, InvalidIP(ip), At );
  checkrec: Record( x, InvalidDN(dn), NSt );
  checkrec: Record( x, InvalidDN(dn), CNAMEt ).

pred isInvalid/1.
clauses
  isInvalid:InvalidIP(ip);
  isInvalid:InvalidDN(dn).

data zero/0. data succ/1.

pred ga/2.
clauses
ga:succ(x),x;
ga:x,y -> ga:succ(x),y.

data true/0.
data mkName/1. data seedRoot/0.
data dnmSeed/0. data cnmSeed/0. data dnmIP/0.

data ERRORID0/0. data ERRORID1/0. data ERRORID2/0.
data ERRORID3/0. data ERRORID4/0. data ERRORID5/0.
data ERRORID6/0. data ERRORID7/0. data ERRORID8/0.
data AAFLAG/0.

data makeSubName/2.
pred isSubName/2.
clauses
  isSubName: x, x;
  isSubName: makeSubName( z, x ), x;
  isSubName: x, y -> isSubName: makeSubName( z, x ), y.

(* Non-overwritabilities in Section 6.3 for A *)
(* query ev: evPoison( At, cached_src_a, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
  ==> ev: evInitCache( Record( cached_src_a, cached_dst_a, At), lva ) & ga: lva, lv.
  ( TRUE: in case, v6,v4 is true, FALSE: in case: v3{E7},v2{E2,E5,E7} )
*)

(* Non-overwritabilities in Section 6.3 for CNAME *)
(* query ev: evPoison( CNAMEt, cached_src_cname, dst, lv,
cached_src_ns, cached_src_a, cached_src_cname, eid ).
(Impossible in every case)
 *)

(* Properties in Section 6.4 *)
(* query ev: evPoison( NSt, x, dst, lv, cached_src_ns,
cached_src_a, cached_src_cname, eid ) ==>
      ev: recursiveQueryStart( y, zone, zone_target )
       & isSubName: x, zone & isSubName: y, zone.

query ev: evPoison( At, x, dst, lv, cached_src_ns,
cached_src_a, cached_src_cname, eid ) ==>
      ev: recursiveQueryStart( y, zone, zone_target )
```

```
       & isSubName: x, zone & isSubName: y, zone.

query ev: evPoison( CNAMEt, x, dst, lv, cached_src_ns,
cached_src_a, cached_src_cname, eid ) ==>
      ev: recursiveQueryStart( y, zone, zone_target )
       & isSubName: x, zone & isSubName: y, zone.
*)

(* Properties in Section 7.1 *)
(* query ev: evPoison( CNAMEt, x, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
  ==>ev: evInitCache( Record( target, validIP, At), lva ).
*)


(* Properties in Section 7.2 *)
(* query ev: evPoison( At, makeSubName(bad, goodAuth), invalidIP, lv,
goodAuth, makeSubName(good, goodAuth), cached_src_cname, eid ) ==>
ev: evInitCache( Record( makeSubName(good, goodAuth), validIP, At), lva ).
ERRORID 0,2,3,5,7
*)

(* Properties in Section 7.3 *)
(* query ev: evPoison( At, cached_src_a, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
  ==> ev: evInitCache( Record( cached_src_a, cached_dst_a, At), lva ) & ga: lva, lv.
   ( FALSE: in case: v3{E7}, v2{E2,E5,E7} )
*)

(* Properties in Section 7.4 *)
(* query ev: evPoison( NSt, cached_src_ns, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
     ==> ev: evInitCache( Record( cached_src_ns, cached_dst_ns, NSt), lvns ) & ga: lvns, lv.
( FALSE in case any lvns lower than v6: v5{E1},v3{E1},v2{E1,E4} )
*)

let processResolver =
   let v0 = succ(zero) in let v1 = succ(v0) in
   let v2 = succ(v1) in let v3 = succ(v2) in
   let v4 = succ(v3) in let v5 = succ(v4) in
   let v6 = succ(v5) in let v7 = succ(v6) in
   let v8 = succ(v7) in

   new  currentState;

   (
      (
      (* Initialize a cache state. Based on this state, Resolver model makes a decision *)
      (* At the initial state, the model caches three records:A,NS,CNAME  *)
         new magicZONE;
         out( net, magicZONE );
         in ( net, ( initNSlabel, initClabel, initAlabel ) );

         let nsRoot =
         Record( initNSlabel, ValidDN( dnmSeed ), NSt ) in (* Make a NS type record with a domain name and specified IP address *)
         let aRoot  =
         Record( initAlabel, ValidIP( dnmIP ), At ) in   (* Make a A type record with a domain name and specified IP address *)
         let cnameRoot =
         Record( initClabel, ValidDN( cnmSeed ), CNAMEt ) in  (* In special case for modeling CNAME record in cache *)

         (* Assert an event declaring cached records with certain trust levels.  *)
(* Before starting the model, User MUST specify the trust level of each record.  *)
         event evInitCache( aRoot, v6 );      (* The trust level of A record can be or 6, 4, 3 or 2 *)
         event evInitCache( nsRoot, v5 );     (* The trust level of NS record can be or 5, 3 or 2 *)
         event evInitCache( cnameRoot, v6 ); (* The trust level of CNAME record can be or 6 or 4 *)

         (* currentState is a private internal channel, It is used for passing cached information to resolving process *)
         out( currentState, ( zeroStage, cnameRoot, aRoot, nsRoot, magicZONE ) )
       )
     | (
         (* Get cache information from init process *)
         in( currentState, ( =zeroStage, cachedCNAMErecord, cachedArecord, cachedNSrecord, mZONE ) );
         (* Get a query request from open channel network.*)
         !in( net, ( makeSubName(inputname, inputzone), inputtype) );

         let input = makeSubName(inputname, inputzone) in

         event revQuery( input , inputtype );   (* Assert an event that a query arrived *)

         (* If there is a cached CNAME type record whose label is same as a A or CNAME type query then, resolution ends *)
         if ( input, inputtype, GetType(cachedCNAMErecord) ) = ( GetSrc( cachedCNAMErecord ), CNAMEt, CNAMEt ) then
         (  event queryResolved( input, inputtype ) )
         else if ( input, inputtype, GetType(cachedCNAMErecord) ) = ( GetSrc( cachedCNAMErecord ), At, CNAMEt ) then
         ( event queryResolved( input, inputtype ) )
         (* If there is a cached NS type record whose label is same as a NS type query then, resolution ends *)
         else if ( input, inputtype ) = ( GetSrc( cachedNSrecord ), NSt ) then
         (  event queryResolved( input, inputtype ) )
         (* If there is no record matching to a given query name and type, a recursive query starts. *)
         else if ( input, inputtype ) <> ( GetSrc( cachedArecord ), GetType( cachedArecord ) ) then
         (
            let srcNSrecord = GetSrc( cachedNSrecord ) in

            (* the label of a received query is subdomain of cached NS, other NS:mZONE or no records *)
```

```
          if input = srcNSrecord then
            out( currentState, ( oneStage, srcNSrecord, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
          else if isSubName: input, srcNSrecord then
            out( currentState, ( oneStage, srcNSrecord, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
          else if input = mZONE then
            out( currentState, ( oneStage, mZONE, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
          else if isSubName: input, mZONE then
            out( currentState, ( oneStage, mZONE, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
          else event resolvingFailed( input )
    ) (* if ( input, inputtype ) <> ( GetSrc( cachedArecord ), GetType( cachedArecord ) )  *)
    else (  event queryResolved( input, inputtype ) )
  )
| (
    !in( currentState, (=oneStage, srcLabelCachedNS, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) );

    new dstLabelCachedNS;
    (* The query must be a subdomain of ZONE determined by a resolver, otherwise it'll fail! *)
    if isSubName: input, srcLabelCachedNS then
    (
       (* Because, there is no data, the model start a recursive query. *)
       out( net, (input, inputtype) );
       event recursiveQueryStart( input, srcLabelCachedNS, dstLabelCachedNS );

       (* And now waiting a response from a AA *)
       in( net, ( Response( =input,
                            b, (*Record( =input, ans_target, =inputtype),*)
                            Record(auth_name, auth_target, =NSt),
                            Record(add_name, add_target, =At) ), aa ) );

       let ans     = b in
       let auth    = Record( auth_name, auth_target, NSt) in
       let add     = Record( add_name, add_target, At) in

       (* Let's check the answer section of the response *)
       if GetSrc(ans) <> nil then
       (
         if (input,inputtype) = (GetSrc(ans),GetType(ans)) then
         (
            if aa = AAFLAG then
            (
               (* Check the validity of reponses. But, in this point, our model resolver has a power to detect the validity of a response. *)
                if checkrec: ans then
                    event evPoison( GetType(ans), GetSrc(ans), GetDst(ans), v6,
                                    GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                    GetSrc(cachedCNAMErecord), ERRORID0 );

                if isSubName: GetSrc(auth), srcLabelCachedNS then
                (
                   if checkrec: auth then
                       event evPoison( NSt, GetSrc(auth), GetDst(auth), v5,
                                       GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                       GetSrc(cachedCNAMErecord), ERRORID1 )
                   else
                   (
                      if GetDst(auth) = GetSrc(add) then
                         if isSubName: GetSrc(add), srcLabelCachedNS then
                           if checkrec: add then
                              event evPoison( GetType(add), GetSrc(add), GetDst(add), v2,
                                       GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                       GetSrc(cachedCNAMErecord), ERRORID2 )
                   )
                ) (* if isSubName: GetSrc(auth), srcLabelCachedNS then  *)
            ) (* if aa = AAFLAG then *)
            else (* if a received reponse is not from an authroritative server *)
            (
               if checkrec: ans then
                  event evPoison( GetType(ans), GetSrc(ans), GetDst(ans), v4,
                                  GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                  GetSrc(cachedCNAMErecord), ERRORID3 );

               if isSubName: GetSrc(auth), srcLabelCachedNS then
               (
                  if checkrec: auth then
                  (   if aa <> AAFLAG then
                        event evPoison( NSt, GetSrc(auth), GetDst(auth), v2,
                                        GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                        GetSrc(cachedCNAMErecord), ERRORID4 )
                  )
                  else
                  (
                     if GetDst(auth) = GetSrc(add) then
                        if isSubName: GetSrc(add), srcLabelCachedNS then
                           if checkrec: add then
                              event evPoison( GetType(add), GetSrc(add), GetDst(add), v2,
                                         GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                         GetSrc(cachedCNAMErecord), ERRORID5 )
                   )
                )
            )(* else aa = AAFLAG *)
         ) (* if input = GetSrc(ans) then *)
```

```
            ) (* if GetSrc(ans) <> nil then *)
            else if GetSrc(ans) = nil then
            (
               if isSubName: input, GetSrc( auth )  then
               (
                  if GetSrc(auth) <> srcLabelCachedNS then
                  (
                     if isSubName: GetSrc( auth ), srcLabelCachedNS then
                     (
                        if checkrec: auth then
                        (
                           event evPoison( NSt, GetSrc(auth), GetDst(auth), v8,
                                                GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                                GetSrc(cachedCNAMErecord), ERRORID6 )
                         )
                         else (* if checkrec: auth then  *)
                         (
                             (* Here, We cache the received NS record because it is valid! *)
                             if isSubName: GetSrc( add ), srcLabelCachedNS then
                                if GetDst(auth) = GetSrc(add) then
                                   if checkrec: add then
                                      event evPoison( GetType(add), GetSrc(add), GetDst(add), v3,
                                                      GetSrc(cachedNSrecord), GetSrc(cachedArecord),
                                                      GetSrc(cachedCNAMErecord), ERRORID7 )
                         ) (* if checkrec: auth then  *)
                     ) (* if isSubName: GetSrc( auth ), srcLabelCachedNS then  *)
                  )  (* if GetSrc(c) <> ZONE *)
                  else (* if GetSrc(c) = ZONE *)
                  (
                      event NSrecordAlreadyCached( GetSrc(auth) )
                  ) (* else for if GetSrc(c) <> ZONE *)
              ) (* isSubName: input, GetSrc(auth) *)
            ) (* if GetSrc(ans) = nil then *)
         ) (* if isSubName: input, srcLabelCachedNS then *)
         else
         (
              (* We have no AA for this query, Therefore, the query is failed. *)
              event resolvingFailed( input )
         )
      )
).

process
 processResolver
```

B Unbound resolver model

```
(************************************************************
 *                                                          *
 *       Unbound DNS resolver verifier                      *
 *                                                          *
 ************************************************************)

param redundantHypElim = beginOnly.
param traceDisplay = long.
param traceBacktracking = false.

fun NSt/0. fun At/0. fun CNAMEt/0.
fun zeroStage/0. fun oneStage/0.

fun ansChID/0. fun authChID/0. fun addChID/0.

data ValidIP/1. data InvalidIP/1.
data ValidDN/1. data InvalidDN/1.
data AA/1. data nil/0. data Record/3.

reduc GetSrc    ( Record( dmnSrc, target, recType ) ) = dmnSrc.
reduc GetDst    ( Record( dmnSrc, target, recType ) ) = target.
reduc GetType    ( Record( dmnSrc, target, recType ) ) = recType.
reduc GetDnDst  ( Record( dmnSrc, ValidDN(target), recType ) ) = target.

data Response/4.
reduc GetAnswer   ( Response( q1, Record(a1,a2,a3), Record(au1,au2,NSt), Record(ad1,ad2,At) ) )
                  = Record(a1,a2,a3).
reduc GetAuth     ( Response( q1, Record(a1,a2,a3), Record(au1,au2,NSt), Record(ad1,ad2,At) ) )
                  = Record(au1,au2,NSt).
reduc GetAdd      ( Response( q1, Record(a1,a2,a3), Record(au1,au2,Nst), Record(ad1,ad2,At) ) )
                  = Record(ad1,ad2,At).

free net.
data emptyset/0.

(* Predicate checking whether it is valid or not *)
pred checkrec/1.
clauses
  checkrec: Record( x, InvalidIP(ip), At );
  checkrec: Record( x, InvalidDN(dn), NSt );
  checkrec: Record( x, InvalidDN(dn), CNAMEt ).

pred isInvalid/1.
clauses
  isInvalid:InvalidIP(ip);
  isInvalid:InvalidDN(dn).

data zero/0. data succ/1.

pred ga/2.
clauses
   ga:succ(x),x;
   ga:x,y -> ga:succ(x),y.

data true/0. data mkName/1.
data seedRoot/0. data dnmSeed/0. data dnmIP/0.

data ERRORID0/0. data ERRORID1/0. data ERRORID2/0.
data ERRORID3/0. data ERRORID4/0. data ERRORID5/0.
data ERRORID6/0. data ERRORID7/0. data ERRORID8/0.
data ERRORID9/0.
data AAFLAG/0. data EXIST/0. data NONEXIST/0.

data makeSubName/2.
pred isSubName/2.
clauses
  isSubName: x, x;
  isSubName: makeSubName( z, x ), x;
  isSubName: x, y -> isSubName: makeSubName( z, x ), y.

(* Non-overwritabilities in Section 6.3 for A *)
(* query ev: evPoison( At, cached_src_a, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
     ==> ev: evInitCache( Record( cached_src_a, cached_dst_a, At), lva ) & ga: lva, lv.
  ( TRUE: in case, v6,v4 is true, FALSE: in case: v3{E7,E8-1,E9-1},v2{E7,E8,E9} )
*)

(* Non-overwritabilities in Section 6.3 for CNAME *)
(*   query ev: evPoison( CNAMEt, cached_src_cname, dst, lv,
                     cached_src_ns, cached_src_a, cached_src_cname, eid ).
   (Impossible in every case)
*)

(* Properties in Section 6.4 *)
(* query ev: evPoison( NSt, x, dst, lv, cached_src_ns,
               cached_src_a, cached_src_cname, eid ) ==>
      ev: recursiveQueryStart( y, zone, zone_target )
         & isSubName: x, zone & isSubName: y, zone.
```

```
   query ev: evPoison( At, x, dst, lv, cached_src_ns,
             cached_src_a, cached_src_cname, eid ) ==>
      ev: recursiveQueryStart( y, zone, zone_target )
         & isSubName: x, zone & isSubName: y, zone.

   query ev: evPoison( CNAMEt, x, dst, lv, cached_src_ns,
             cached_src_a, cached_src_cname, eid ) ==>
      ev: recursiveQueryStart( y, zone, zone_target )
         & isSubName: x, zone & isSubName: y, zone.
*)

(* Properties in Section 7.1 *)
(* query ev: evPoison( CNAMEt, x, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
     ==>   ev: evInitCache( Record( target, validIP, At), lva ).
   ERRORID8, ERRORID7, ERRORID6
*)

(* Properties in Section 7.2 *)
(* query ev: evPoison( At, makeSubName(bad, goodAuth), invalidIP, lv,
                goodAuth, makeSubName(good, goodAuth), cached_src_cname, eid ) ==>
         ev: evInitCache( Record( makeSubName(good, goodAuth), validIP, At), lva ).
   ERRORID 0, 1, 9, 8, 7
*)

(* Properties in Section 7.3 *)
(* query ev: evPoison( At, cached_src_a, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
     ==> ev: evInitCache( Record( cached_src_a, cached_dst_a, At), lva ) & ga: lva, lv.
  ( TRUE: in case, v6,v4 is true, FALSE: in case: v3{E7,E8-1,E9-1},v2{E7,E8,E9} )
*)

(* Properties in Section 7.4 *)
(* query ev: evPoison( NSt, cached_src_ns, dst, lv, cached_src_ns, cached_src_a, cached_src_cname, eid )
     ==> ev: evInitCache( Record( cached_src_ns, cached_dst_ns, NSt), lvns ) & ga: lvns, lv.
   ( FALSE in case any lvns lower than v6: v5{E4,E5-1},v3{E4,E5-1},v2{E4,E5} )
*)

let processCC =
   let v0 = zero in   let v1 = succ(v0) in
   let v2 = succ(v1) in   let v3 = succ(v2) in
   let v4 = succ(v3) in   let v5 = succ(v4) in
   let v6 = succ(v5) in   let v7 = succ(v6) in
   let v8 = succ(v7) in

   new currentState;
   new ansChannel;
   new authChannel;
   new addChannel;

   event StartResolver( currentState );

   (
      (
      (* Initialize a cache state. Based on this state, Resolver model makes a decision *)
      (* At the initial state, the model caches three records:A,NS,CNAME  *)
         new magicZONE;

         out( net, magicZONE );
         in ( net, ( initNSlabel, initClabel, initAlabel ) );

         let nsRoot =
         Record( initNSlabel, ValidDN( dnmSeed ), NSt ) in (* Make a NS type record with a domain name and specified IP address *)
         let aRoot  =
         Record( initAlabel, ValidIP( dnmIP ), At ) in     (* Make a A type record with a domain name and specified IP address *)
         let cnameRoot =
         Record( initClabel, ValidDN( cnmSeed ), CNAMEt ) in  (* In special case for modeling CNAME record in cache *)

         (* Assert an event declaring cached records with certain trust levels.  *)
(* Before starting the model, User MUST specify the trust level of each record.  *)
         event evInitCache( aRoot, v6 );     (* The trust level of A record can be or 6, 4, 3 or 2 *)
         event evInitCache( nsRoot, v5 );    (* The trust level of NS record can be or 5, 3 or 2 *)
         event evInitCache( cnameRoot, v6 ); (* The trust level of CNAME record can be or 6 or 4 *)

         (* currentState is a private internal channel, It is used for passing cached information to resolving process *)
         out( currentState, ( zeroStage, cnameRoot, aRoot, nsRoot, magicZONE ) )
     )
   | (
         (* Get cache information from init process *)
         in( currentState, ( =zeroStage, cachedCNAMErecord, cachedArecord, cachedNSrecord, mZONE ) );
         (* Get a query request from open channel network.*)
         !in( net, ( makeSubName(inputname, inputzone), inputtype) );

         let input = makeSubName(inputname, inputzone) in

         event revQuery( input , inputtype );                             (* Assert an event that a query arrived *)

         (* If there is a cached CNAME type record whose label is same as a A or CNAME type query then, resolution ends *)
         if ( input, inputtype, GetType(cachedCNAMErecord) ) = ( GetSrc( cachedCNAMErecord ), CNAMEt, CNAMEt ) then
         (  event queryResolved( input, inputtype ) )
         else if ( input, inputtype, GetType(cachedCNAMErecord) ) = ( GetSrc( cachedCNAMErecord ), At, CNAMEt ) then
         (   event queryResolved( input, inputtype ) )
```

```
    (* If there is a cached NS type record whose label is same as a NS type query then, resolution ends *)
    else if ( input, inputtype ) = ( GetSrc( cachedNSrecord ), NSt ) then
    (  event queryResolved( input, inputtype ) )
    (* If there is no record matching to a given query name and type, a recursive query starts. *)
    else if ( input, inputtype ) <> ( GetSrc( cachedArecord ), GetType( cachedArecord ) ) then
    (
       let srcNSrecord = GetSrc( cachedNSrecord ) in

       if input = srcNSrecord then
          out( currentState, ( oneStage, srcNSrecord, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
       else if isSubName: input, srcNSrecord then
          out( currentState, ( oneStage, srcNSrecord, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
       else if input = mZONE then
          out( currentState, ( oneStage, mZONE, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
       else if isSubName: input, mZONE then
          out( currentState, ( oneStage, mZONE, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
       else event resolvingFailed( input )
    ) (* if ( input, inputtype ) <> ( GetSrc( cachedArecord ), GetType( cachedArecord ) )  *)
    else
    (  event queryResolved( input, inputtype ) )
 )
| (
    !in( currentState, (=oneStage, srcLabelCachedNS, input, inputtype, cachedNSrecord, cachedArecord, cachedCNAMErecord ) );

    new dstLabelCachedNS;

    (* The query must be a subdomain name of zone determined by a resolver, otherwise it'll fail! *)
    if isSubName: input, srcLabelCachedNS then
    (
       let bailiwickZone   = srcLabelCachedNS in

       (* Because, there is no data, the model start a recursive query. *)
       out( net, (input, inputtype) );

       event recursiveQueryStart( input, srcLabelCachedNS, dstLabelCachedNS );

       (* And now waiting a response from a AA *)
       in( net, ( Response( =input, b,
                         Record(auth_name, auth_target, =NSt),
                         Record(add_name, add_target, =At) ), aa ) );

       let ans     = b in
       let auth    = Record( auth_name, auth_target, NSt) in
       let add     = Record( add_name, add_target, At) in

       (
          (
             out( ansChannel, ( ansChID , input, inputtype, ans , aa,
                               bailiwickZone, cachedNSrecord, cachedArecord, cachedCNAMErecord ) )
          )
       |  (
             if ans <> nil then
                out( authChannel, (authChID , input, inputtype, EXIST, auth, aa,
                                  bailiwickZone, cachedNSrecord, cachedArecord, cachedCNAMErecord) )
          )
       |  (
             if ans <> nil then
                out( addChannel,  (addChID  , input, inputtype, EXIST, auth, add, aa,
                                  bailiwickZone, cachedNSrecord, cachedArecord, cachedCNAMErecord) )
          )
       |  (
             if ans = nil then
                out( authChannel, (authChID , input, inputtype, NONEXIST , auth , aa,
                                  bailiwickZone, cachedNSrecord, cachedArecord, cachedCNAMErecord) )
          )
       |  (
             if ans = nil then
                out( addChannel,  (addChID  , input, inputtype, NONEXIST , auth , add, aa,
                                  bailiwickZone, cachedNSrecord, cachedArecord, cachedCNAMErecord) )
          )
       )

    )   (* if isSubName: input, srcLabelCachedNS then *)
 )
| (
    (* Internal Answer Channel *)
    !in ( ansChannel, (=ansChID, inputR, inputtypeR, ansR, aaR, bailiwickZoneR,
                     cachedNSrecordR, cachedArecordR, cachedCNAMErecordR ) );

    if GetType( ansR ) <> NSt then
    (
       if ansR <> nil then
       (
          if ( inputR, inputtypeR ) = ( GetSrc( ansR ), GetType( ansR ) ) then
          (
             if isSubName:  GetSrc(ansR), bailiwickZoneR then
             (
                if aaR = AAFLAG then
                (
                      if checkrec: ansR then
```

```
                        event evPoison( GetType(ansR), GetSrc(ansR), GetDst(ansR), v6,
                              GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID0 )
                     )
                     else if checkrec: ansR then
                        event evPoison( GetType(ansR), GetSrc(ansR), GetDst(ansR), v4,
                           GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID1 )
                  )
               )
               else if ( inputR, inputtypeR, GetType( ansR ) ) = ( GetSrc( ansR ), At, CNAMEt ) then
               (
                  if isSubName:  GetSrc(ansR), bailiwickZoneR then
                  (
                     if aaR = AAFLAG then
                     (
                        if checkrec: ansR then
                        (
                           event evPoison( CNAMEt, GetSrc(ansR), GetDst(ansR), v6,
                              GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID2 )
                        )
                     )
                     else if checkrec: ansR then
                        event evPoison( CNAMEt, GetSrc(ansR), GetDst(ansR), v4,
                              GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID3 )
                  )
               ) (* else if ( inputR, inputtypeR, GetType( ansR ) ) = ( GetSrc( ansR ), At, CNAMEt ) then  *)
            ) (* if ( inputR, inputtypeR ) = ( GetSrc( ansR ), GetType( ansR ) ) then  *)
         ) (* if GetType( ansR ) <> NSt then *)
       )
   | (
        (* Auth Channel *)
         !in ( authChannel, (=authChID, inputR, inputtypeR, ansR, authR, aaR, bailiwickZoneR,
                           cachedNSrecordR, cachedArecordR, cachedCNAMErecordR ) );

         if isSubName: GetSrc(authR), bailiwickZoneR then
         (
            if (ansR,aaR) = (EXIST,AAFLAG) then
           (
               if checkrec: authR then (* trust_auth_AA *)
                event evPoison( NSt, GetSrc(authR), GetDst(authR), v5,
                           GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID4 )
           ) (* if (ansR,aaR) = (EXIST,AAFLAG) then *)
           else if GetSrc(authR) = bailiwickZoneR then
           (
              if checkrec: authR then (* trust_auth_AA *)
              (
                 if aaR = AAFLAG then
                   event evPoison( NSt, GetSrc(authR), GetDst(authR), v5,
                              GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID5 )
                 else
                   event evPoison( NSt, GetSrc(authR), GetDst(authR), v2,
                              GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID5 )
              )
           )
           else if isSubName: inputR, GetSrc(authR) then
           (
              if checkrec: authR then (* trust_auth_AA *)
              (
                 if aaR = AAFLAG then
                     event evPoison( NSt, GetSrc(authR), GetDst(authR), v5,
                              GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID6 )
                 else
                    event evPoison( NSt, GetSrc(authR), GetDst(authR), v2,
                           GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID6 )
              )
           )
           else
           (
              event cachingAuthfail( authR )
           )
        ) (* if isSubName: GetSrc(authR), bailiwickZoneR then *)
     )
   | (    (* Add Channel *)
       !in ( addChannel, (=addChID, inputR, inputtypeR, ansR, authR, addR, aaR, bailiwickZoneR,
                       cachedNSrecordR, cachedArecordR, cachedCNAMErecordR ) );

     if isSubName: GetSrc(authR), bailiwickZoneR then
         if isSubName: GetSrc(addR), bailiwickZoneR then
            if GetSrc(addR) = GetDst(authR) then
      (
         event guideEvent3();

         if (ansR,aaR) = (EXIST,AAFLAG) then
        (
            if checkrec: addR then
              event evPoison( At, GetSrc(addR), GetDst(addR), v3,
                           GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID7 )
        )

        (* else if isSubName: bailiwickZoneR, GetSrc(authR) then  *)
        else if GetSrc(authR) = bailiwickZoneR then
```

```
          (
             if checkrec: addR then
             (
                if aaR = AAFLAG then
                   event evPoison( At, GetSrc(addR), GetDst(addR), v3,
                            GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID8 )
                else
                   event evPoison( At, GetSrc(addR), GetDst(addR), v2,
                            GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID8 )
             )
          )

           (* else if isSubName: GetSrc(addR), bailiwickZoneR then *)
          else if isSubName: inputR, GetSrc(authR) then
          (
             if inputR <> GetSrc(authR) then
             (
                if checkrec: addR then
                (
                   if aaR = AAFLAG then
                        event evPoison( At, GetSrc(addR), GetDst(addR), v3,
                            GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID9 )
                   else
                      event evPoison( At, GetSrc(addR), GetDst(addR), v2,
                            GetSrc(cachedNSrecordR), GetSrc(cachedArecordR), GetSrc(cachedCNAMErecordR), ERRORID9 )
                )
             )
             else event cachingGLUEfail( addR )
          )
          else
          (
             event cachingGLUEfail( addR )
          )
       )
    )
).

process
    processCC
```