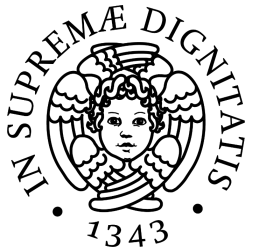
Analysis of the DNS Cache-Poisoning Attack using PRISM Model Checker



Academic Year 2025/2026

Giacomo Lombardi
Nicolò Zarulli

Table of Contents

Table of Contents

Table of Contents

Modelling the Kaminsky Attack on PRISM

- Transition Systems & Continuous Time Markov Chain
- PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port
- Property verification and experiments
- DNSSEC solution, why not enough?

Table of Contents

Modelling DNS-CPM Rℓ1 on PRISM

- DNS-CPM Mitigation
- DNS-CPM Mitigation (Rl1)
- PRISM Model of Kaminsky DNS Cache-Poisoning Attack with DNS-CPM Mitigation
- Property verification and experiments

Table of Contents

Modelling DNS-CPM Rℓ2 on PRISM

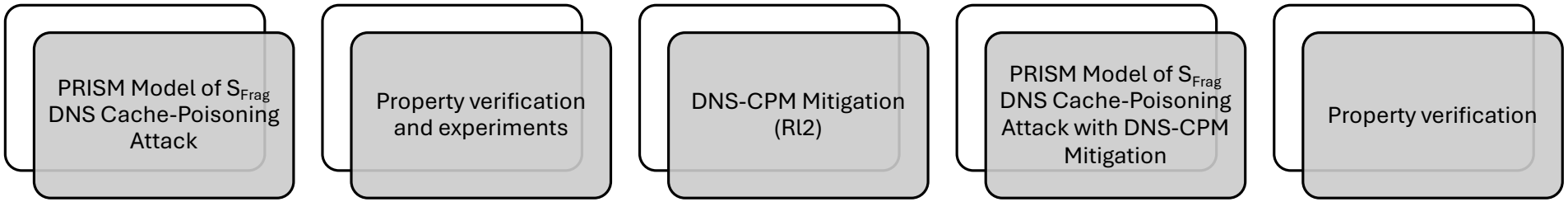
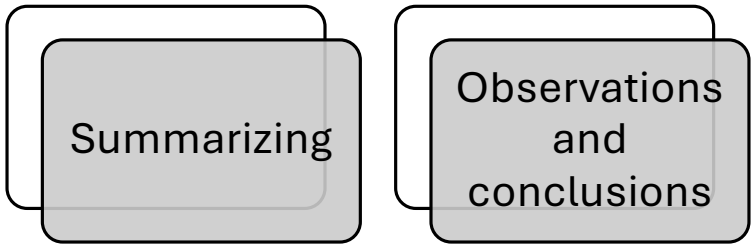


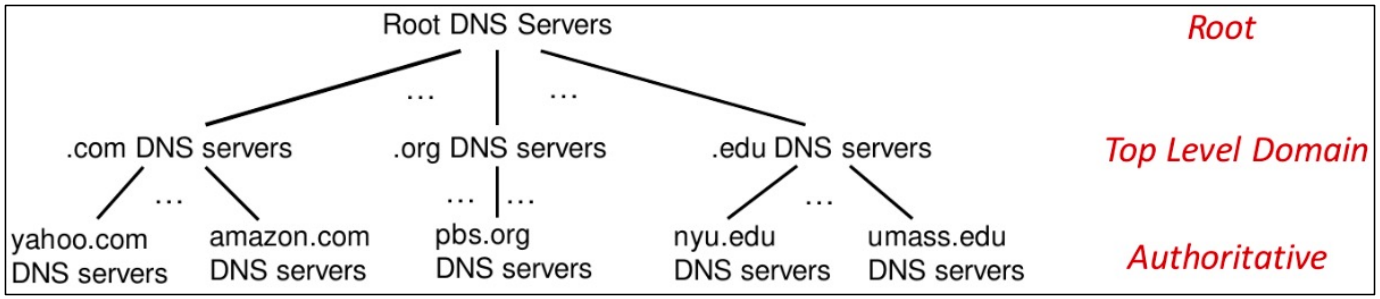
Table of Contents

Conclusion



Domain Name System (DNS)

- DNS is a distributed database with hierarchical structure



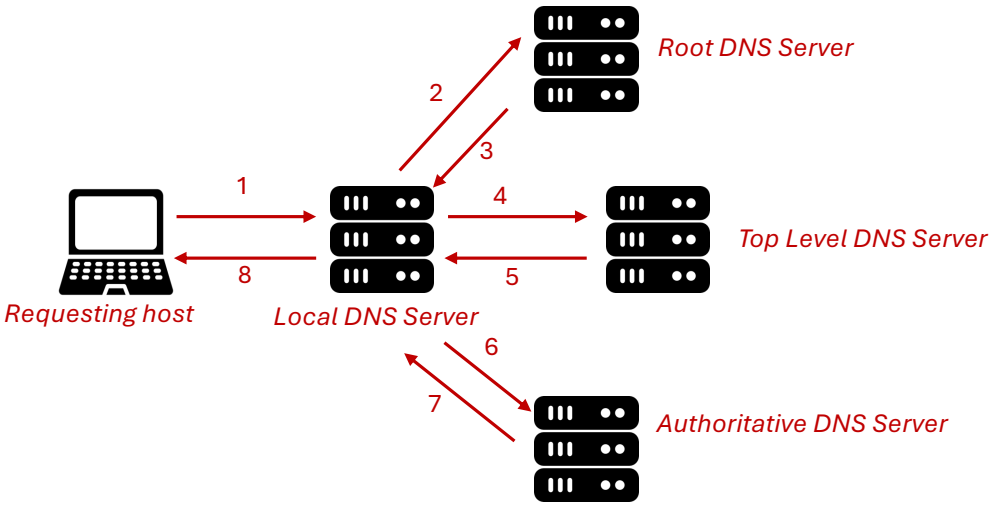
- An entry in the database is called ***Resource Record***: <*Name, Value, Type, Class, TTL*>
  - *Name* and *Value* depend on the *Type*
  - Most famous types are: *"A"*, corresponding to the mapping of name-IPv4, and *"AAAA"*, corresponding to the mapping name-IPv6
  - *TTL* specifies how long the RR is valid (can be kept in cache)

- DNS is an application layer protocol

- Most DNS traffic is over ***UDP*** since DNS is just a request/replay protocol

Domain Name System (DNS)

- DNS name resolution approach: ***Iterative***



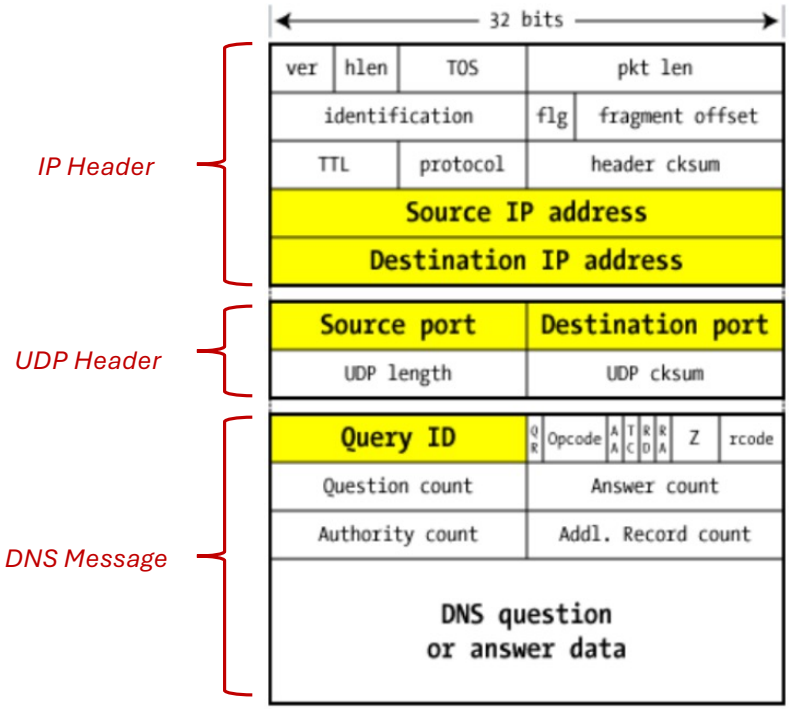
Example

1. *Host → Resolver: IP of www.google.com*
2. *Resolver → Root DNS Server: IP of www.google.com*
3. *Root DNS Server → Resolver: IP of TLD DNS Server (.com DNS Server)*
4. *Resolver → TLD DNS Server: IP of www.google.com*
5. *TLD DNS Server → Resolver: IP of Authoritative DNS Server (google.com DNS Server)*
6. *Resolver → google.com DNS Server: IP of www.google.com*
7. *Google.com DNS Server → Resolver: Authoritative Answer (AA) containing the IP*
8. *Resolver → Host: forwards AA*

- *Local DNS Server = Resolver* → extracts information from DNS Servers in response to clients' requests
- Resolver saves *Domain Name-IP* associations in its ***cache***

Domain Name System (DNS)

- DNS packet
- DNS message encapsulated in an UDP header
- Which is encapsulated in an IP header
- DNS query:
  - **Source IP** = IP of the client sending the DNS query
  - **Destination IP** = IP of the server
  - **Source port** = chosen by the OS
  - **Destination port** = 53
  - Addresses are exchanged in the response
  - **Query ID**: unique identifier created in the query packet; the responding server uses the same Query ID



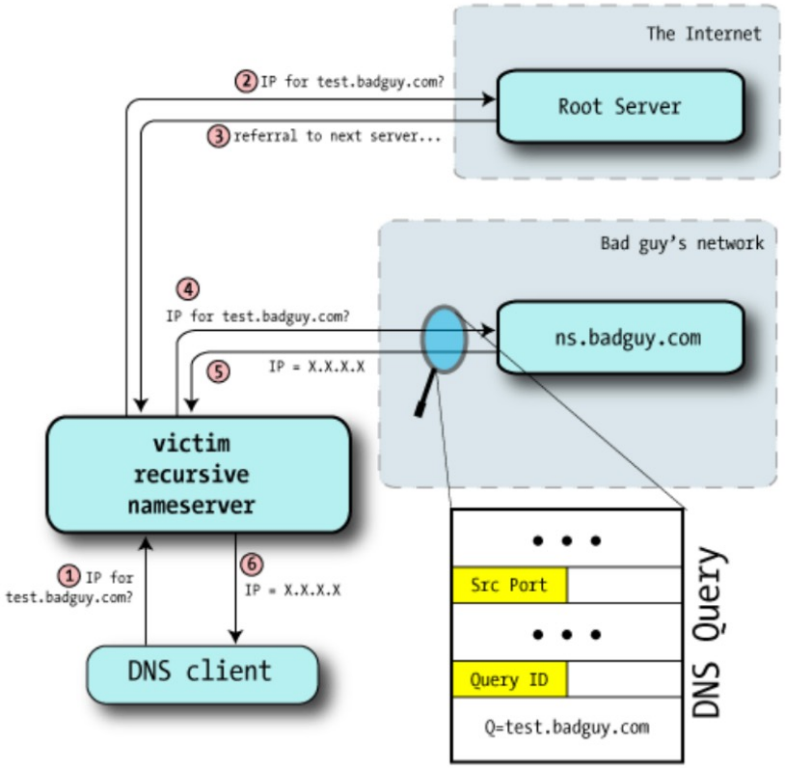
DNS Cache-Poisoning Attacks

Idea: the attacker inserts wrong information in the cache of the resolver, so that if some user makes a certain DNS query will receive a wrong response.

Intuitively: When the resolver sends a request to an Authoritative DNS Server, if the attacker can forge a formally correct DNS response, the resolver will accept it as correct if it arrives before the legitimate one from the Authoritative DNS Server.

Formally correct DNS response: The formally correct DNS response has the same Query ID of the DNS request and Destination Port equal to the Source Port of the DNS request

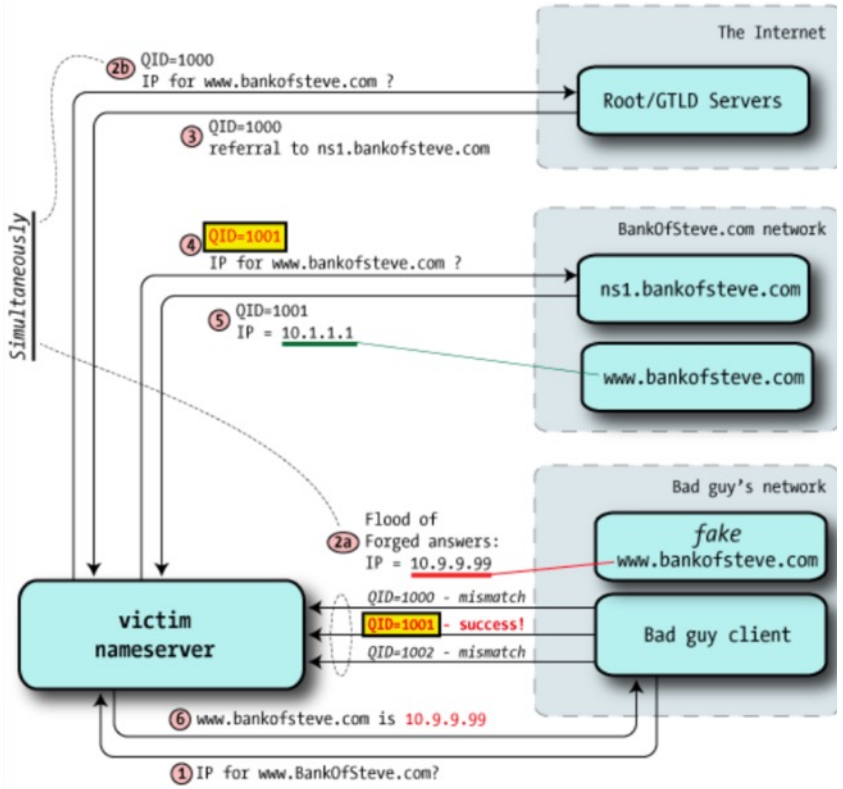
Original vulnerabilities:
- Resolver always use the *same Source Port*
- *Incremental Query ID* for each new request made by the resolver



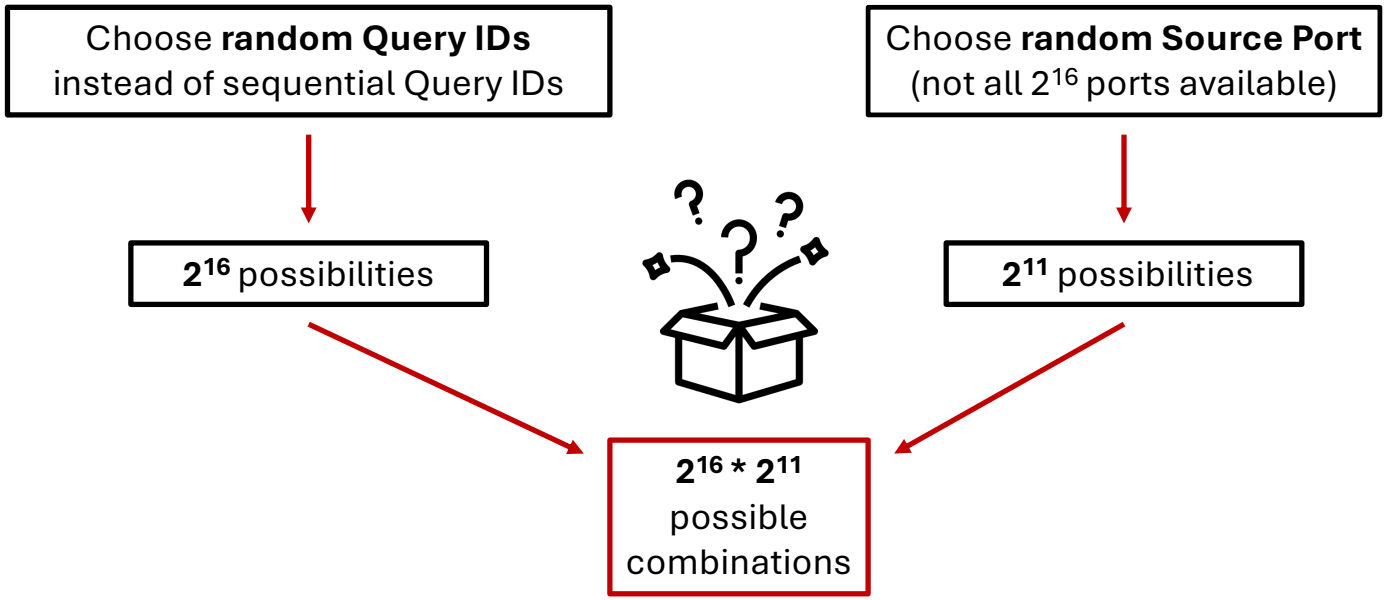
DNS Cache-Poisoning Attacks

General attack steps:

1. Suppose the attacker has obtained the Source Port used by the resolver to make queries and a reference value for Query ID.
2. Trick the resolver to query the Authoritative DNS Server of the *target domain D,* expecting a response with the *IP address of D*.
3. While the resolver is expecting a response from authoritative DNS Server, attacker provides it with a forged response with the source IP of the Authoritative DNS Server.
4. For the forged response to be accepted by the resolver it must have the correct Destination Port number, Query ID, and domain *D*, in addition to the correct IP address of Authoritative DNS Server.



Mitigation: Randomization of Query ID and Source Port



Types of DNS Cache-Poisoning Attacks

***S type* (Kaminsky attack):** Attacker floods the resolver with many guesses trying different combinations of Destination Port number and Query ID, hoping that one of them will have the correct numbers. It has more or less 2^{32} possible combinations.

***S_{Frag} type* (Fragmentation attack):** When fragmentation of IP packets is used, second fragments lack UDP and DNS headers, meaning no Destination Port and no Query ID have to be guessed. The only security check is the 16-bit IPID (2^{16} possibilities).
Attacker first sends a series of forged *2nd fragment,* with different IPIDs, containing the mapping it wants to be cached. Then, it tricks the resolver to query Authoritative DNS Server with a query whose response size requires fragmentation.
The 2nd fragments wait in the OS reassembly buffer and when the legitimate 1st fragment arrives, the OS reassembles it with the malicious 2nd fragment if the IPID matches

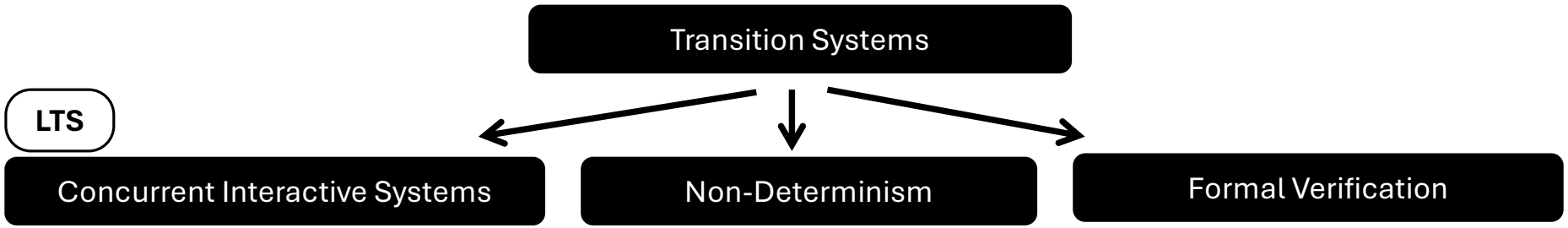
***S_{OoB} type* (Out-of-Bailwick):** This is a variant of Kaminsky attack, in which the attacker includes in the forged response an OoB record with the mapping it wishes to insert into the cache (e.g. a mapping for *example.net* while the original query was for *bankofsteve.com*. The *.net* domain is out of bailwick for *.com*). Due to inadequate validation, the resolver may cache the additional OoB mappings.

Transition Systems

The DNS Cache Poisoning Attacks works on the basis of a → **Time Based Race Condition**

Intruder vs. Authoritative Name Server

We want to model the environment in which an attacker can pose a DNS Cache Threat and being able the trace all the possible paths that may lead to an attack. Given therefore that we are interested in studying whether a Victim DNS Resolver reaches a bad state, hence “cache poisoned”, ***Transition Systems*** offer us the underlying structure to do just that.



We need to model an environment made of different **independent components** that may **perform actions synchronizing** with each other and the environment

To model choices and uncertainty

We use PRISM both to model the system and to carry out a process of Model Checking

Continuous Time Markov Chain

The DNS Cache Poisoning Attacks work on the basis of a → **Time Based Race Condition**

Intruder vs. Authoritative Name Server

The suitable choice for us to model the behavior of a DNS Cache Poisoning Attack is the ***Continuous Time Markov Chain (CTMC).*** Why a CTMC instead of a DTMC?

Real Time vs Step-Based Time

The DTMC progesses in fixed steps, and at each step one transition occurs. Almost like the ticking of a clock.

Problem: DNS packets won't wait for a clock tick but may be sent at any given time. In a CTMC a transition can happen at any moment (random intervals).

Attacker Race Problem

The DTMC struggles with races, hence evaluating the next transition when two things are happening at the same time but at two different speeds.

The CTMC uses the concept of ***Rates*** to evaluate the "next transition". In the case that two events are scheduled, the one with the higher rate is statistically more likely to happen!

Continuous Time Markov Chain

By using a CTMC we are then able to model:

The Race Condition

Cache can be either «poisoned» or «safe»

Winner Takes All Philosophy: The first packet to arrive and be accepted determines the state (poisoned vs. safe).

In a CTMC the fastest transition determines the next state

Real World Metrics and Attack Parameters

DNS Attack Traffic and Real Network Traffic are defined by **Rates (events per second).**

CTMCs use a Transition Rate Matrix where rates are based on **real-world parameters** (Attacker Guess Rate vs DNS Server Response Arrival Rate).

The model can account for Server Load where higher traffic ***slows*** down the legitimate server, ***increasing*** the intruder's window of opportunity

Queueing and Delays

Waiting Times

Time spent in a state is governed by a negative exponential distribution, simulating variable network latency and processing delays.

Vulnerability Window

The model captures the specific duration the victim server is "waiting" for a reply, which is the exact window the attacker must exploit

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

Modules

Intruder Machine
Intruder's PC that requests resolution of a URL

Victim Resolver
Local DNS Server that responds to the client's requests

Root DNS Server
DNS Server that knows the IP address of the Authoritative DNS Server

Domain Server
Authoritative DNS Server of the requested domain

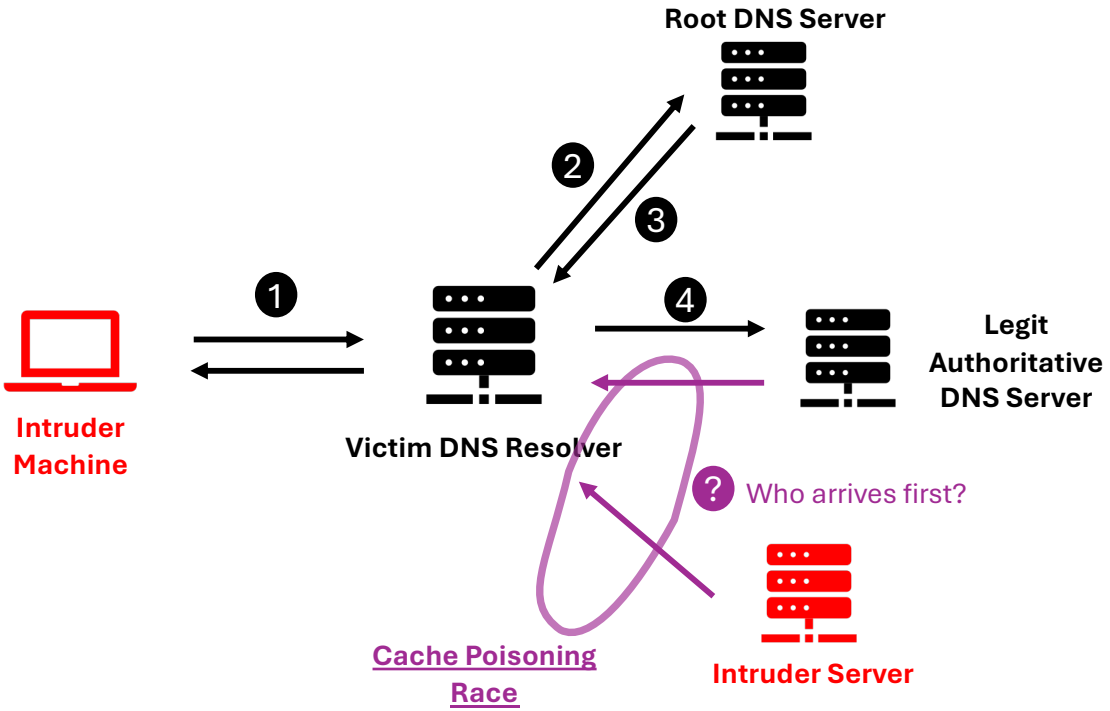
Intruder Server
Intruder's DNS Server trying to guess correct Query ID (and Port number)

Final states of the model

1) A correct guess has been made → Cache-Poisoning Attack success

2) All requests from Intruder Machine answered correctly

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port



PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

Model parameters

Number of URL requests: Number of URL requests attacker sends to resolver (i.e. different attacks)

Max queue: Total number of requests attacker can send to resolver

Popularity: Rate at which the TTL of the resolver's cache entry for the domain the attacker is trying to poisoning has a positive value

Query ID range Number of possible Query IDs (set by default to $2^{16} = 65536$)

Port ID range: Range of Port numbers for the purpose of implementing source-port randomization

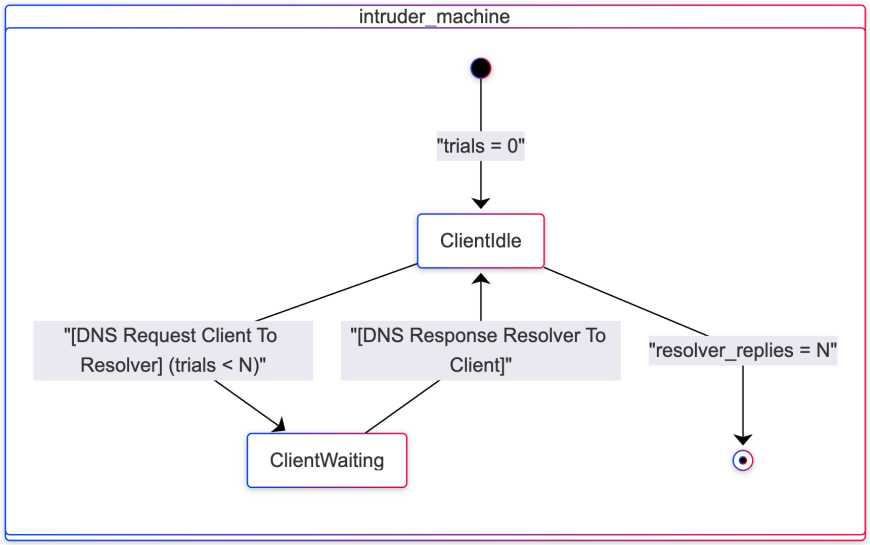
Guess: Rate at which the attacker server sends fake responses to resolver

Requests rate: Rate at which the attacker sends requests to resolver

Authoritative DNS Workload: Workload on the Authoritative DNS Server for the "attacked" domain, related to traffic different than the one from victim resolver

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

```
module intruder_machine
        trials : [0..NUMBER_OF_URL_REQUESTS] init 0;
        resolver_replies : [0..NUMBER_OF_URL_REQUESTS] init 0;

        // -- Transitions
        // 1. Send Request: Initiates resolution.
        [DNS_Request_Client_To_Resolver] trials < NUMBER_OF_URL_REQUESTS -> requests_rate : (trials' = trials + 1);

        // 2. Receive Reply: Marks end of query cycle.
        [DNS_Response_Resolver_To_Client] resolver_replies < trials -> (resolver_replies' = resolver_replies + 1);

        // 3. Termination
        [] resolver_replies = NUMBER_OF_URL_REQUESTS -> true;
endmodule
```



PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

```
module victim_resolver
        ttl : [0..2] init 2;
        client_queue : [0.. MAX_QUEUE] init 0;
        root_queue : [0.. MAX_QUEUE] init 0;
        domain_queue : [0.. MAX_QUEUE] init 0;
        responses_queue : [0.. MAX_QUEUE] init 0;
        query_to_root_server: bool init false;
        query_to_domain_server: bool init false;
        answer_from_domain_received: bool init false;
        correct_guess: bool init false;

        // -- Transitions
        // 1. Client Request (Merged Hit/Miss Logic)
        [DNS_Request_Client_To_Resolver] (client_queue < MAX_QUEUE) & (responses_queue < MAX_QUEUE) & (root_queue < MAX_QUEUE) ->
                // Branch 1: Cache Hit
                (popularity / 10) : (client_queue' = client_queue + 1) & (ttl' = 1) & (responses_queue' = responses_queue + 1)
                +
                // Branch 2: Cache Miss
                (1 - (popularity / 10)) : (client_queue' = client_queue + 1) & (ttl' = 0) & (root_queue' = root_queue + 1);

        // 2. Query Root
        [DNS_Request_Resolver_To_Root] (client_queue > 0) & (root_queue > 0)
                -> (query_to_root_server' = true);

        // 3. Root Responds -> Move to Domain Queue
        [DNS_Response_Root_To_Resolver] (root_queue > 0) & (domain_queue < MAX_QUEUE)
                -> (root_queue' = root_queue - 1) & (domain_queue' = domain_queue + 1) & (query_to_root_server' = false);

        // 4. Query Domain (Opens Vulnerability Window)
        [DNS_Request_Resolver_To_Domain] domain_queue > 0 -> (query_to_domain_server' = true);

        // 5. Domain Responds (Closes Window)
        [DNS_Response_Domain_To_Resolver] (domain_queue > 0) & (responses_queue < MAX_QUEUE) & (correct_guess = false)
                -> (domain_queue' = domain_queue - 1) & (responses_queue' = responses_queue + 1) & (query_to_domain_server' = false)
                        & (answer_from_domain_received' = true);

        // 6. Reply to Client
        [DNS_Response_Resolver_To_Client] (responses_queue > 0) & (client_queue > 0)
                -> (client_queue' = client_queue - 1) & (responses_queue' = responses_queue - 1);
```

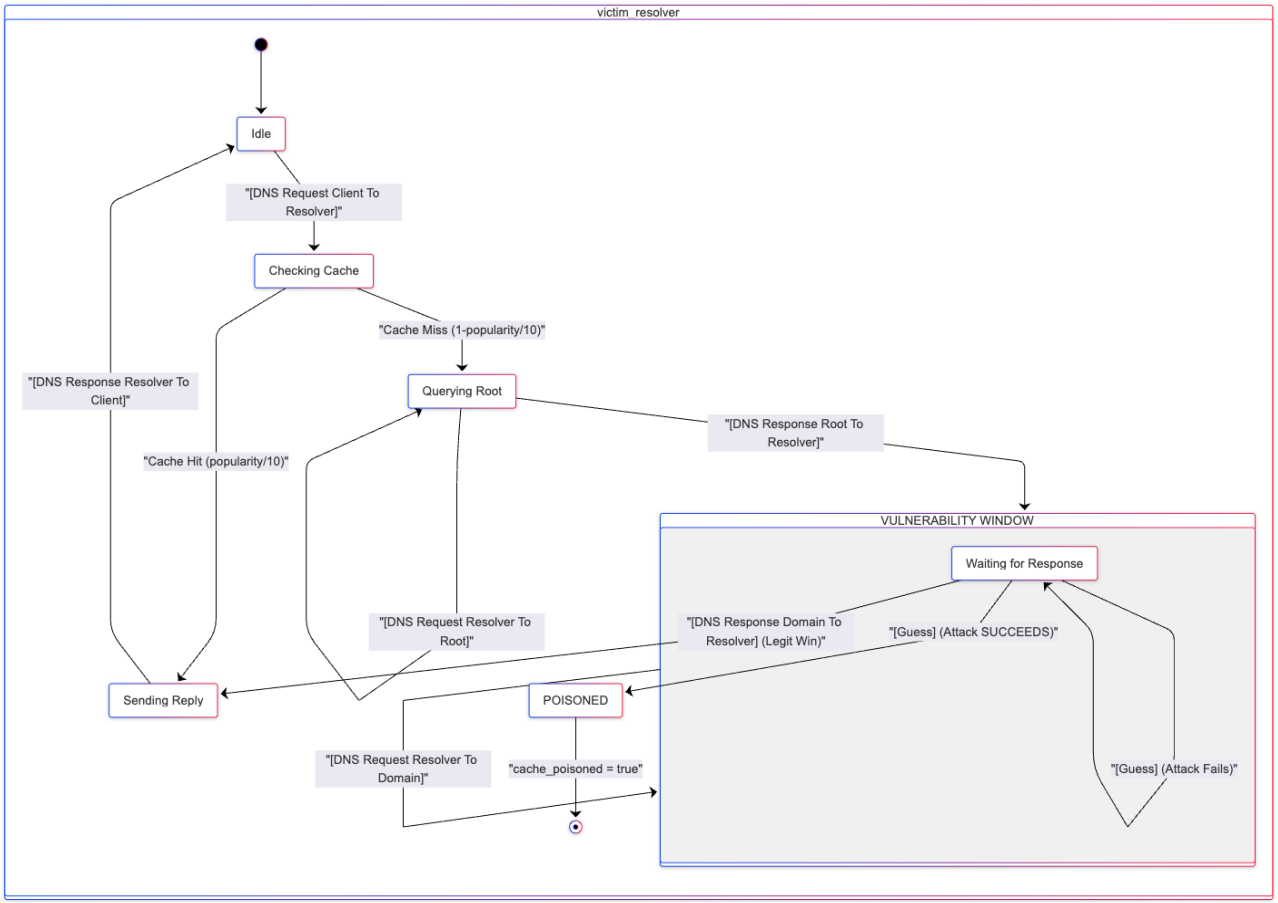
PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

```
        // --- Race Condition (Attack) ---
        [Guess] (correct_guess = false) & (query_to_domain_server = true) ->
                1 / (query_id_range * port_id_range) : (correct_guess'=true)
                +
                ((query_id_range * port_id_range) - 1) / (query_id_range * port_id_range): (correct_guess'=false);

        // Sink State
        [] cache_poisoned = true -> true;
endmodule
```

Rate with which the attacker makes a correct guess:

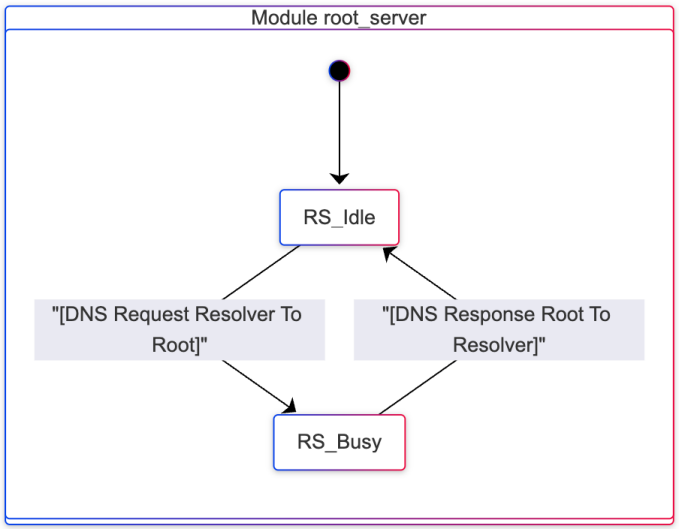
$$guess * \frac{1}{query\_id\_range * port\_id\_range} = guess * \frac{1}{2^{16} * port\_id\_range}$$

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port



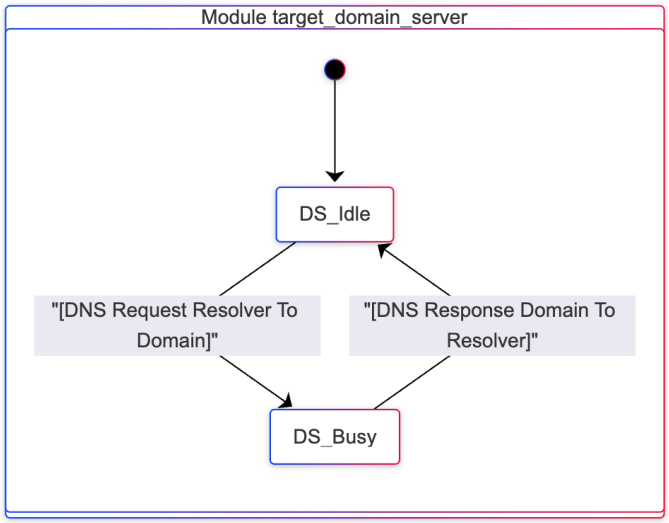
PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

```
module root_server
        root_state : [0..1] init 0;

        // Transitions

        // 1. Receive Request: A query arrives from the Victim Resolver.
        [DNS_Request_Resolver_To_Root] root_state = 0 -> (root_state' = 1);

        // 2. Send Reply: The Root Server immediately responds, moving the resolver to the next step (Domain Server).
        [DNS_Response_Root_To_Resolver] root_state = 1 -> (root_state' = 0);
endmodule
```



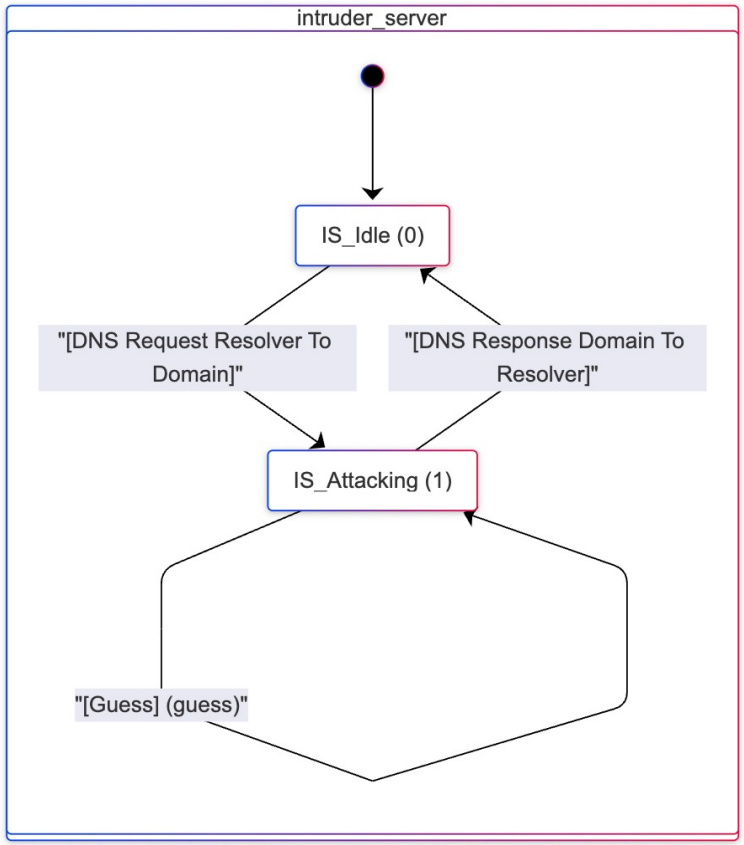
PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

```
module target_domain_server
        // -- States
        domain_state : [0..1] init 0;

        // -- Transitions
        // 1. Receive Request: The Victim Resolver asks "Where is example.com?".
        // This opens the 'Race Window'. The Intruder starts guessing NOW.
        [DNS_Request_Resolver_To_Domain] domain_state = 0 -> (domain_state' = 1);

        // 2. Legitimate Response (The "Good" Ending):
        // The server processes the specific request from our Victim Resolver.
        // This transition closes the 'Race Window' (Intruder stops guessing).
        [DNS_Response_Domain_To_Resolver] domain_state = 1 -> 1/(authoritative_dns_workload) : (domain_state' = 0);
endmodule
```



PRISM Model of Kaminsky DNS Cache-Poisoning Attack with Randomization of Query ID and Source Port

```
module intruder_server
        is_state : [0..1] init 0;

        // -- Transitions
        // 1. Start Attack: Triggered when the Victim Resolver queries the Domain Server.
        // The 'Race Window' is now open.
        [DNS_Request_Resolver_To_Domain] is_state = 0 -> (is_state' = 1);

        // 2. Firing Loop (The "Machine Gun"):
        // Fires guesses at rate 'guess'.
        // The success/failure logic is handled by the Victim Resolver module via probabilistic split.
        [Guess] is_state = 1 -> guess : (is_state' = 1);

        // 3. Stop Attack: Triggered when the Victim Resolver receives the legitimate answer.
        // The 'Race Window' is closed. The attack failed for this specific query.
        [DNS_Response_Domain_To_Resolver] is_state = 1 -> (is_state' = 0);
endmodule
```



Property verification and experiments

P = ? [F cache_poisoned]

Property Verification

Property:
P=? [F cache_poisoned]

Defined constants:
NUMBER_OF_URL_REQUESTS=1,popularity=0,port_id_range=1,guess=300,authoritative_dns_workload=150

Method:
Verification

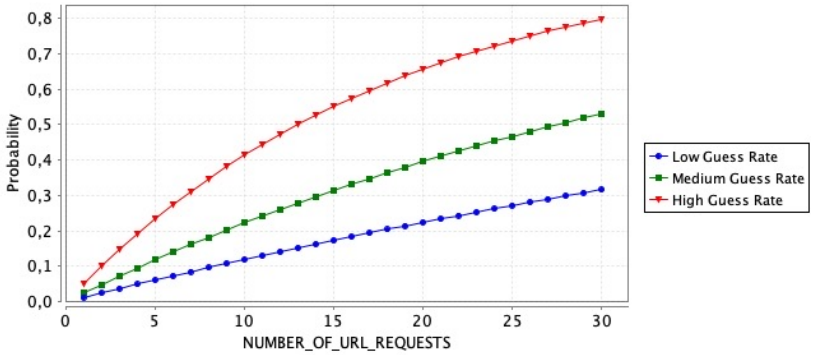
Result (probability):
0.4071071867990663 (value in the initial state)

With only Query ID randomization (no Port randomization), a high guess rate, and a high workload on the Authoritative DNS Server, the attacker has a ***40%*** of ***probability of poisoning the cache*** of the resolver

Property verification and experiments

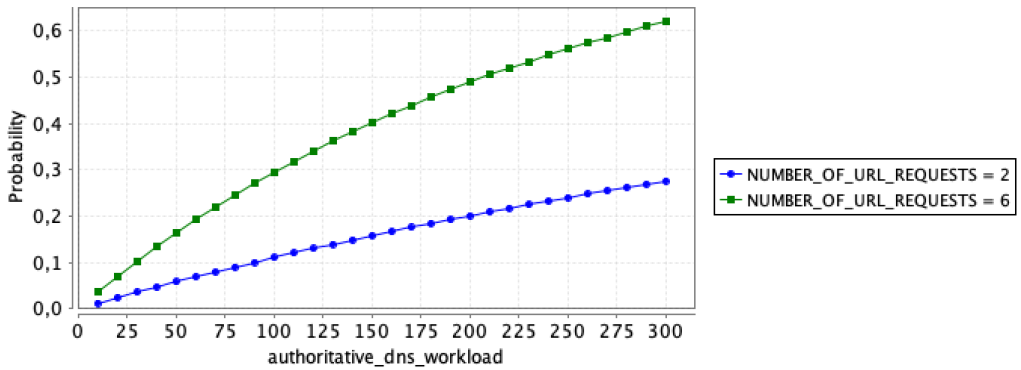
P = ? [F cache_poisoned]

Experiments

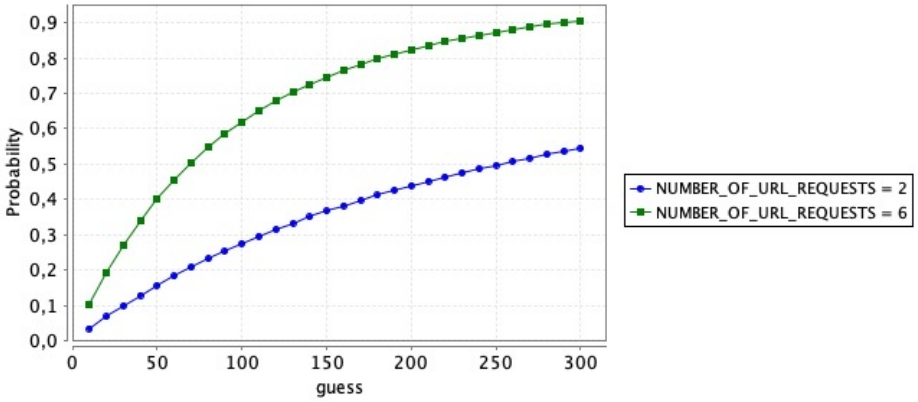


For three values of parameter *guess* (low=30, medium=60, high=130), we vary *number_of_url_requests* from 1 to 30; *authoritative_dns_workload* =50 and *port_id_range*=1. As expected, the more URL-resolution requests there are for the target domain, the higher the probability of attack success

Property verification and experiments

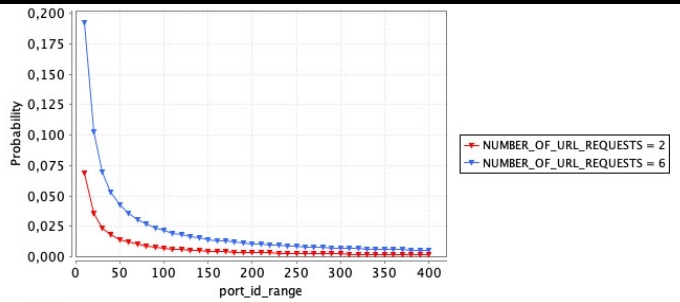


For two values of parameter *number_of_url_requests* (*2 and 6*), we vary *authoritative_dns_workload* from 1 to 300; *guess*=50 and *port_id_range*=1.
As expected, the higher the workload on Authoritative DNS Server, the higher the probability of attack success.

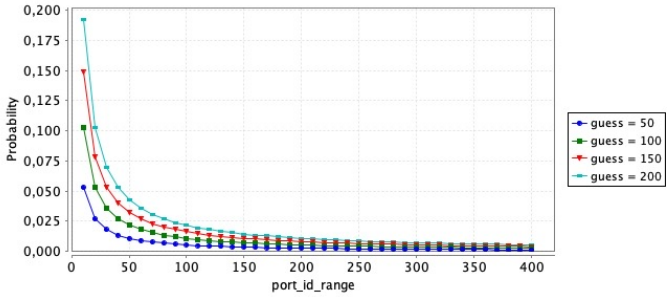


For two values of parameter *number_of_url_requests* (2 and 6), we vary *guess* from 1 to 300; *authoritative_dns_workload* =150 and *port_id_range*=1.
As expected, increasing rate *guess* increases the probability of attack success

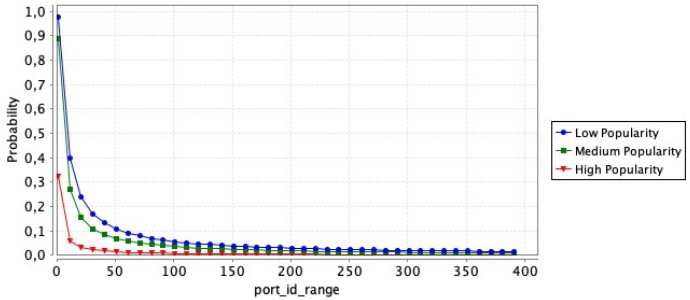
Property verification and experiments



For two values of parameter *number_of_url_requests* (2 and 6), we vary *port_id_range* from 1 to 400; *authoritative_dns_workload* =150 and *guess*=200.

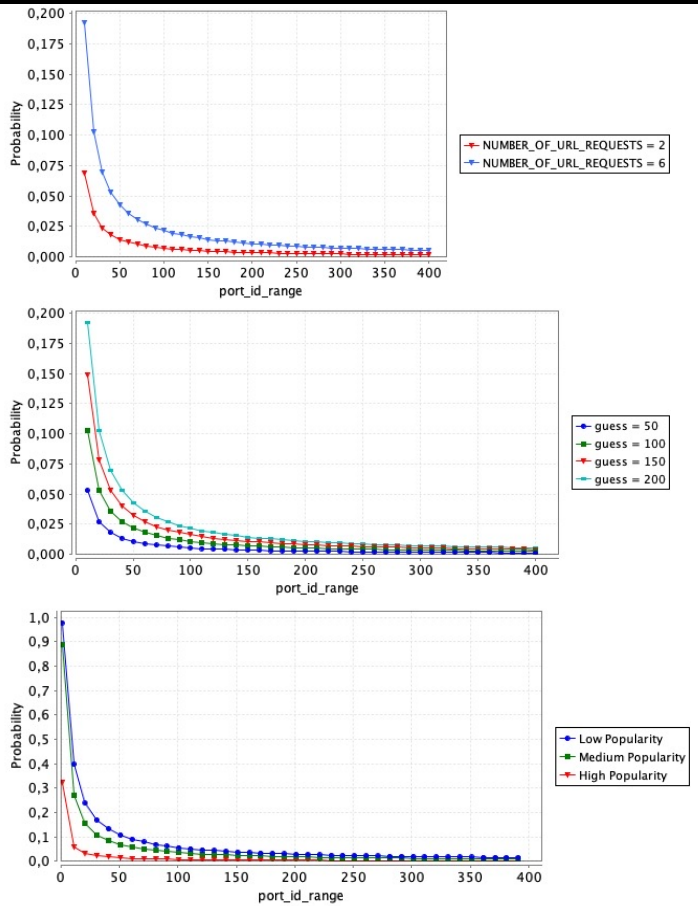


Considering four values of parameter *guess* (50, 100, 150, and 200), we vary *port_id_range* from 1 to 400; *number_of_url_requests*=6 and *authoritative_dns_workload* =150.



For three values of parameter *popularity* (low=2, medium=5, high=9), we vary *port_id_range* from 1 to 400; *authoritative_dns_workload* =300 and *number_of_url_requests*=8.

Property verification and experiments



The probability of attack success decreases nonlinearly as the value of *port_id_range* increases, because the attacker has to guess both Query ID and Port Number

One Possible Solution: DNSSEC

DNS Security Extension (DNSSEC) protocol was designed to eliminate DNS Cache-Poisoning Attacks exploiting ***digital signatures***.

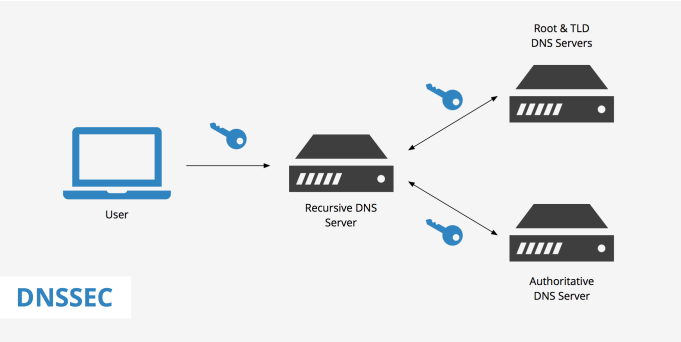
Guarantees *Integrity* and *Authentication* of DNS RR

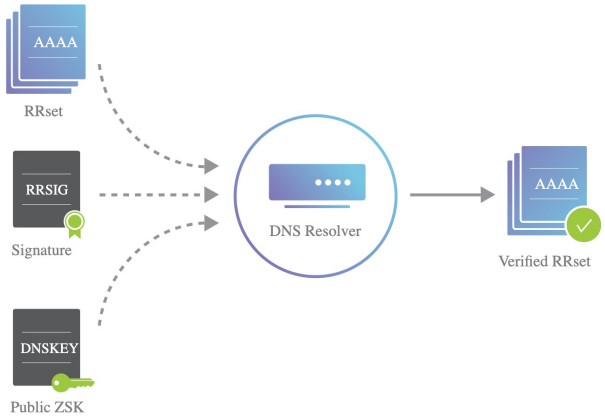
Protects from Spoofing, MIM Attacks and Cache Poisoning

Establishes a «***Chain of Trust***» based on Zone Authority that authenticates DNS Resource Records from their origin, given that DNS resolves queries with an *iterative* approach.

New DNS Resource Records: RRSIG, DNSKEY, DS, NSEC, CDNSKEY

Signature of a RR Set: Multiple RR Authenticated at the same time. Usage of Asymmetric Zone Keys (ZSK).





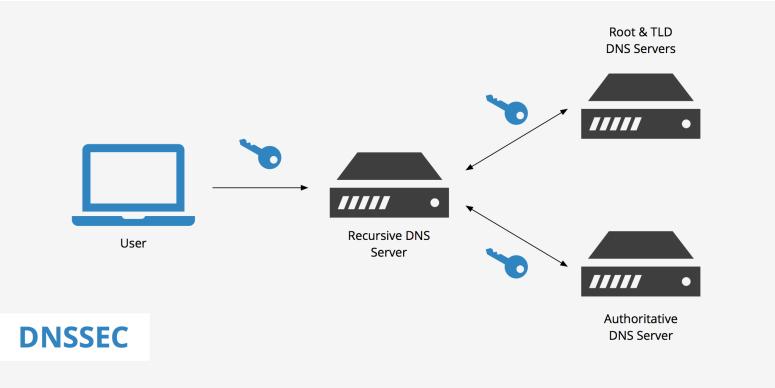
DNSSEC solution: why not enough?

DNS Security Extension (DNSSEC) protocol was designed to eliminate DNS Cache-Poisoning Attacks exploiting ***digital signatures***.

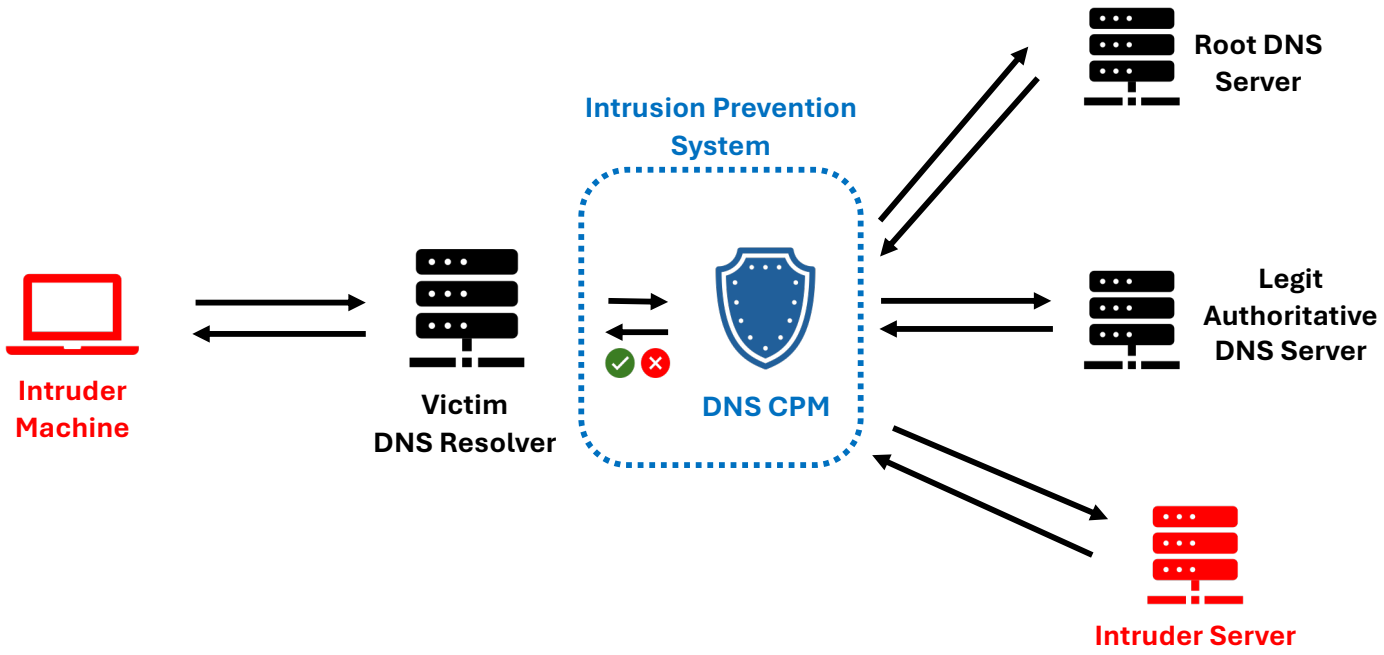
However, we cannot only rely on DNSSEC for the following reasons:

1) Its adoption is ***limited*** (<30% of DNS Servers over the Internet)
2) It heavily ***depends on*** Authoritative DNS Servers implementation

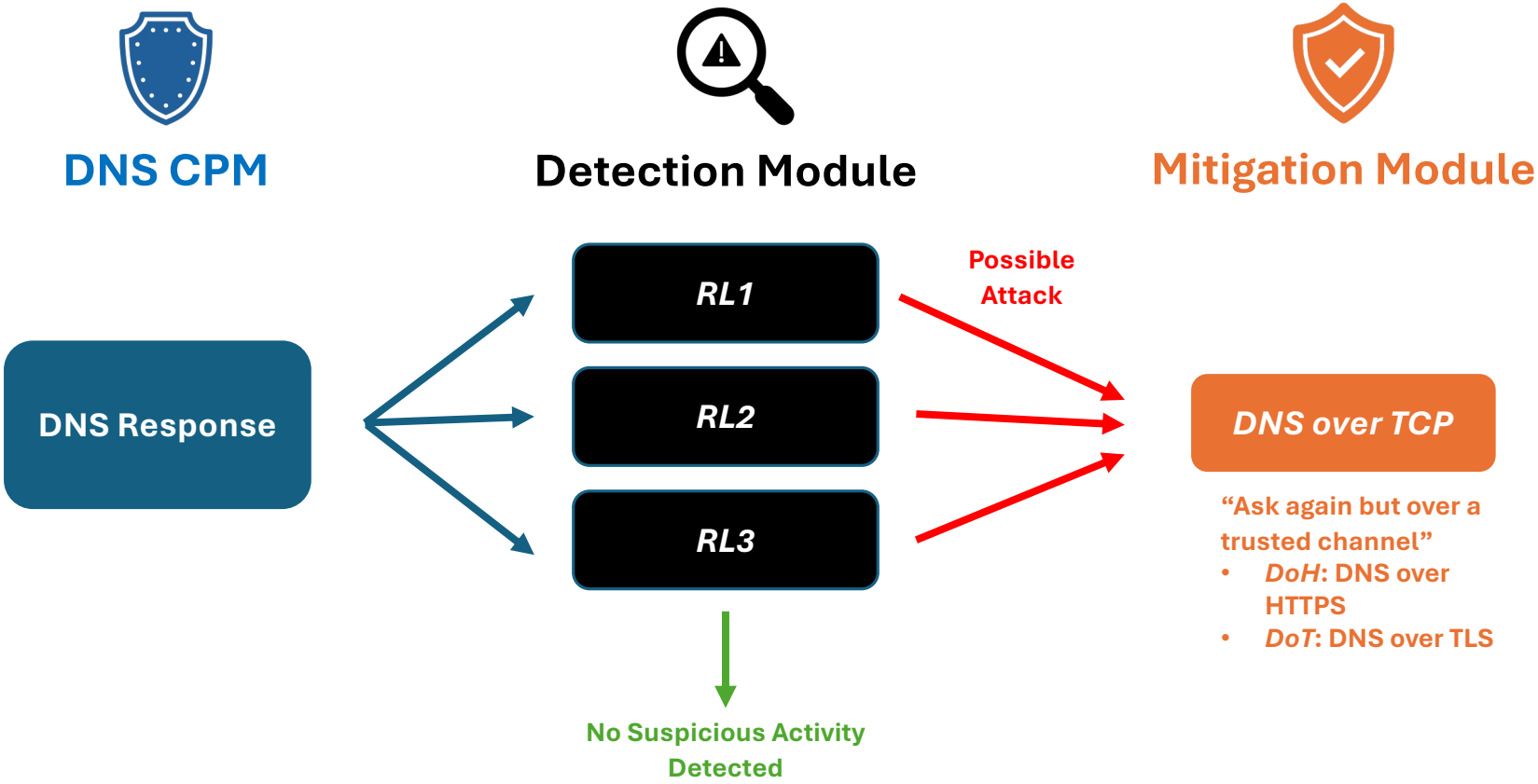
A possible **solution** that moves protection to the resolver is **DNS-CPM**, which acts as an *Intrusion Prevention System (IPS)* against DNS Cache-Poisoning Attacks



DNS-CPM


Root DNS Server
Intrusion Prevention System
Legit Authoritative DNS Server
Intruder Machine
Victim DNS Resolver
DNS CPM
Intruder Server

DNS-CPM Detection & Mitigation



DNS-CPM Detection & Mitigation

Detection Module

Applies different rules to identify different types of DNS Cache-Poisoning Attacks

Rℓ1 against *S attacks*, ***Rℓ2*** against S_{Frag} *attacks*, and ***Rℓ3*** against S_{OoB} *attacks*

When responses are considered suspicious, they are flagged as **potential poisoning attack** and passed to the Mitigation Module

Mitigation Module

When potential poisoning attack responses arrive, the TC flag of those packets is set to true.

When a packet with **TC = true** arrives to the resolver, query/response conversation is moved from DNS over UDP to **DNS over TCP**.

Even if a correct guess arrives to the resolver (matching both Query ID and Port number), it opens a TCP session with authoritative DNS Server, thus preventing the attack

DNS-CPM Mitigation (Rℓ1)

Detection Module (Rℓ1)

Monitor the number of DNS response packets for a URL request that differ from each other only in one common field, i.e. destination port or Query ID. To do so, we considered the **Count-Min Sketch (CMS)** algorithm (error rate = 0.01%)

When the counted number crosses a *threshold* (e.g. 5 packets), the responses are flagged as a **potential poisoning attack** and passed to the Mitigation Module

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with DNS-CPM Mitigation

Modules

Intruder Machine
Intruder's PC that requests resolution of a URL

Victim Resolver
Local DNS Server that responds to the client's requests

Root DNS Server
DNS Server that knows the IP address of the Authoritative DNS Server

Domain Server
Authoritative DNS Server of the requested domain

Intruder Server
Intruder's DNS Server trying to guess correct Query ID (and Port number)

+

DNS-CPM Module
IPS that implements Detection with R1 and Mitigation modules

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with DNS-CPM Mitigation

Additional model parameters

5 ***Tau***

Threshold of DNS response packets for a URL request with identical fields, except for Query ID/Port number. When the counter in the DNS-CPM Module exceeds Tau, the mitigation mechanism is activated

CMS error rate

Considering the way CMS works, this represents the rate of other responses that arrive to the victim resolver that are wrongly associated to the request made by the attacker

↓

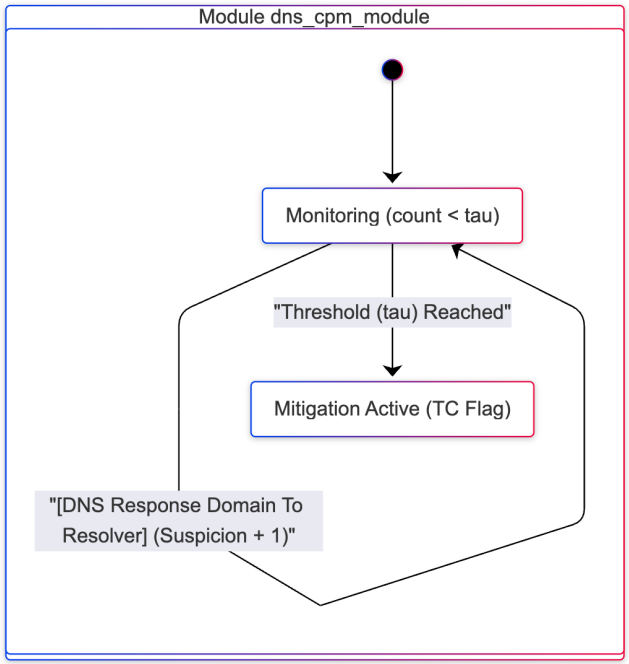
CMS uses *hash functions* to verify if response packets differ only for one field value. Collisions' probability is low, but not zero.

Benign noise rate

Network traffic arriving to the resolver other than the one related to the attacker.
It represents legitimate responses to requests of other clients.

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with DNS-CPM Mitigation

```
module dns_cpm_module
        suspicious_count : [0..tau] init 0;     // Count-Min Sketch counter
        detection_signal : bool init false;     // Signal to the Mitigation Module (True if threshold reached)

        // 1. Detection of a Possible Attack
        [Guess] suspicious_count < tau ->
            (suspicious_count' = suspicious_count + 1) &
            (detection_signal' = (suspicious_count + 1 >= tau));

        // 2. Detection of Benign Noise: Normal traffic towards the victim resolver. If a hash collision occurs, increment counter.
        [Benign_Response] suspicious_count < tau ->
                // Case 1: Collision occurs
                cms_error_rate * ((benign_noise_rate - 1) / benign_noise_rate) :
                        (suspicious_count' = suspicious_count + 1) & (detection_signal' = (suspicious_count + 1 >= tau))
                +
                // Case 2: No Collision
                (1 - cms_error_rate) * (1 / benign_noise_rate): (suspicious_count' = suspicious_count) & (detection_signal' = detection_signal);

        // 3. Detection of a Legitimate Domain Response: Counts towards the threshold (Same Domain)
        [DNS_Response_Domain_To_Resolver] suspicious_count < tau ->
                (suspicious_count' = suspicious_count + 1) &
                (detection_signal' = (suspicious_count + 1 >= tau));
endmodule
```

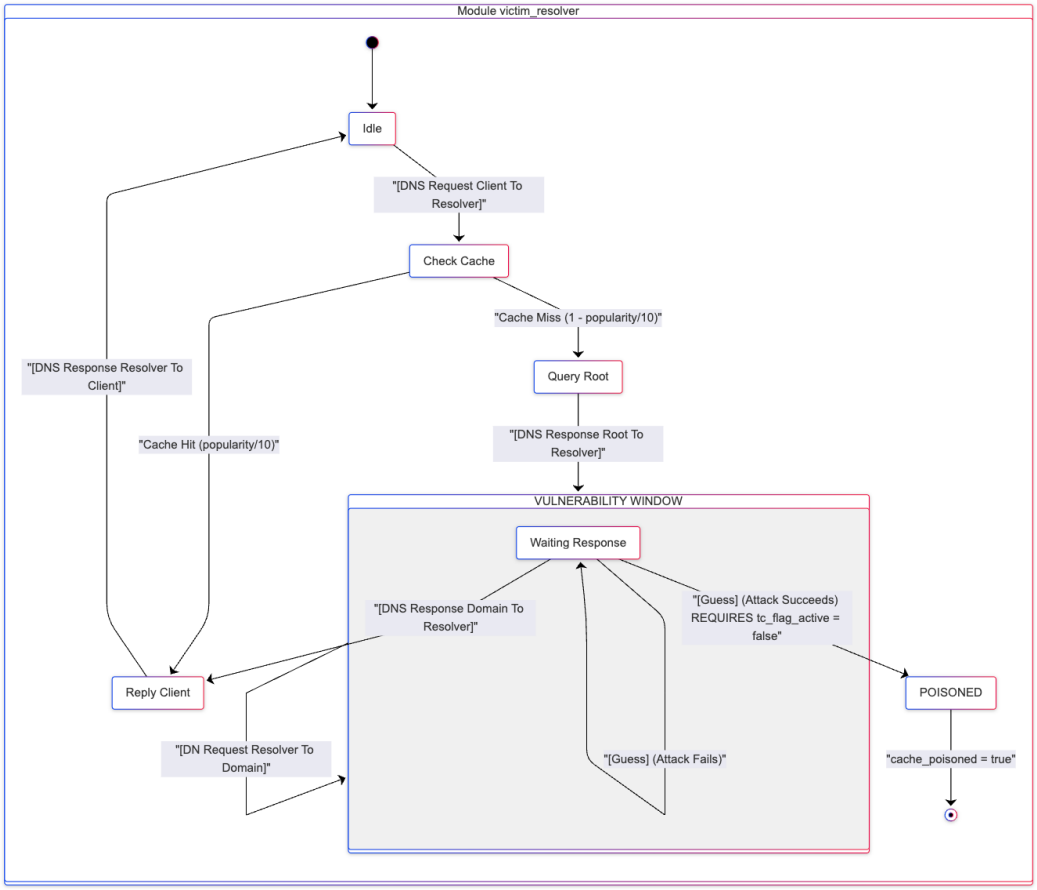


PRISM Model of Kaminsky DNS Cache-Poisoning Attack with DNS-CPM Mitigation

```
module victim_resolver
        ttl : [0..2] init 2; // 0=Miss, 1=Hit, 2=Init
        client_queue : [0..MAX_QUEUE] init 0;
        root_queue : [0..MAX_QUEUE] init 0;
        domain_queue : [0..MAX_QUEUE] init 0;
        responses_queue : [0..MAX_QUEUE] init 0;

        // -- Variables
        query_to_root_server: bool init false;
        query_to_domain_server: bool init false;
        answer_from_domain_received: bool init false;
        correct_guess: bool init false;

        // -- Transitions
        // 1. Client Request (Merged Hit/Miss Logic)
        [DNS_Request_Client_To_Resolver] (client_queue < MAX_QUEUE) & (responses_queue < MAX_QUEUE) & (root_queue < MAX_QUEUE) ->
                // Branch 1: Cache Hit
                (popularity / 10) : (client_queue' = client_queue + 1) & (ttl' = 1) & (responses_queue' = responses_queue + 1)
                +
                // Branch 2: Cache Miss
                (1 - (popularity / 10)) : (client_queue' = client_queue + 1) & (ttl' = 0) & (root_queue' = root_queue + 1);

        // 2. Query Root
        [DNS_Request_Resolver_To_Root] (client_queue > 0) & (root_queue > 0)
                -> (query_to_root_server' = true);

        // 3. Root Responds -> Move to Domain Queue
        [DNS_Response_Root_To_Resolver] (root_queue > 0) & (domain_queue < MAX_QUEUE)
                -> (root_queue' = root_queue - 1) & (domain_queue' = domain_queue + 1) & (query_to_root_server' = false);

        // 4. Query Domain (Opens Vulnerability Window)
        [DNS_Request_Resolver_To_Domain] domain_queue > 0 -> (query_to_domain_server' = true);

        // 5. Domain Responds (Closes Window)
        [DNS_Response_Domain_To_Resolver] (domain_queue > 0) & (responses_queue < MAX_QUEUE) & (correct_guess = false)
                -> (domain_queue' = domain_queue - 1) & (responses_queue' = responses_queue + 1) & (query_to_domain_server' = false)
                        & (answer_from_domain_received' = true);

        // 6. Reply to Client
        [DNS_Response_Resolver_To_Client] (responses_queue > 0) & (client_queue > 0) -> (client_queue' = client_queue - 1)
                & (responses_queue' = responses_queue - 1);
        // -- Attack Race Condition
        [Guess] (correct_guess = false) & (query_to_domain_server = true) & (tc_flag_active = false) ->
                1 / (query_id_range * port_id_range) : (correct_guess'=true)
                +
                ((query_id_range * port_id_range) - 1) / (query_id_range * port_id_range): (correct_guess'=false);

        // -- Sink State
        [] cache_poisoned = true -> true;
endmodule
```

Difference with previous Victim Resolver module

PRISM Model of Kaminsky DNS Cache-Poisoning Attack with DNS-CPM Mitigation



Property verification and experiments

P = ? [F cache_poisoned]

Property Verification

Property:
P=? [F cache_poisoned]

Defined constants:
NUMBER_OF_URL_REQUESTS=1,popularity=0,port_id_range=1,guess=300,authoritative_dns_workload=150,benign_noise_rate=1000

Method:
Verification

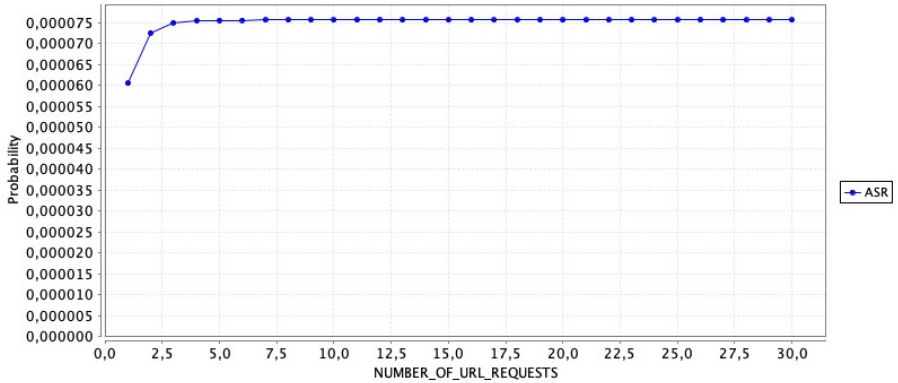
Result (probability):
7.567437471068218E-5 (value in the initial state)

With only Query ID randomization (no Port randomization), a high guess rate, and a high workload on the Authoritative DNS Server, the attacker has a ***0.0076%*** of ***probability of poisoning the cache*** of the resolver

Property verification and experiments

P = ? [F cache_poisoned]

Experiments

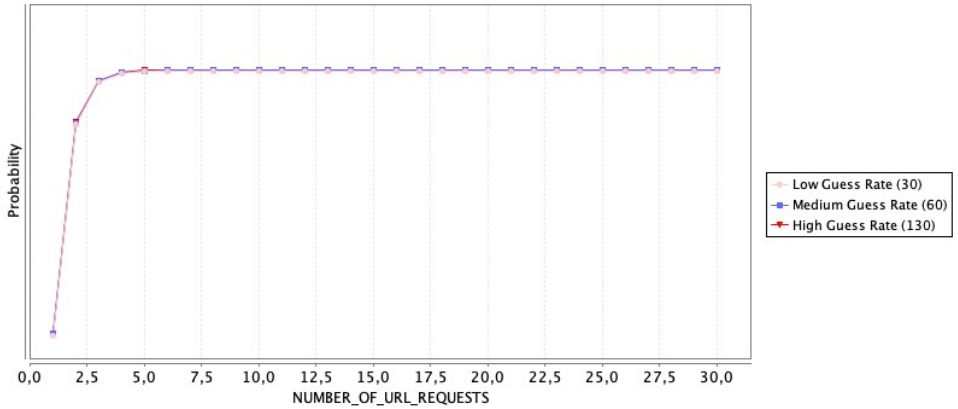


We plot the probability of attack success as a function of *number_of_url_requests*, with *popularity*=2, *port_id_range*=1, *guess*=300, *authoritative_dns_workload*=35, *tau*=5, and *benign_noise_rate*=1000.

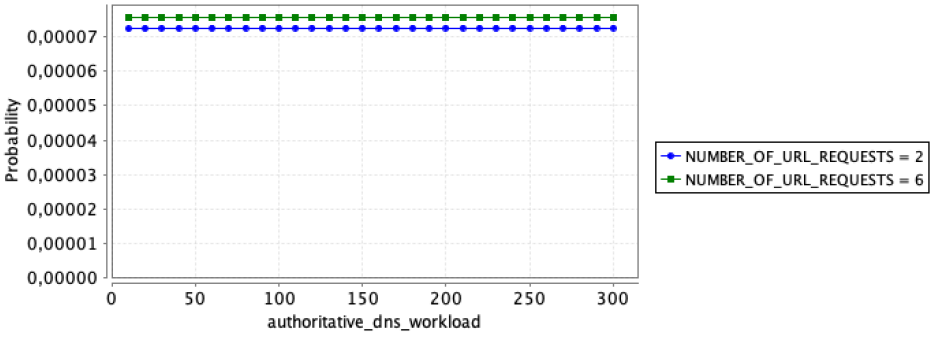
The probability of attack success changes for values of *number_of_url_requests* varying from 1 to 5. After that, probability of success is almost constant.

Property verification and experiments

Replication of the experiments done for model without DNS-CPM Mitigation

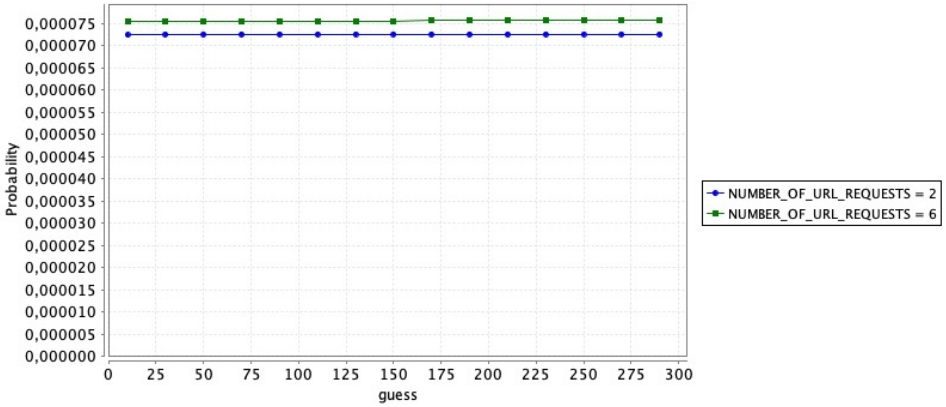


For three values of parameter *guess* (low=30, medium=60, high=130), we vary *number_of_url_requests* from 1 to 30; *authoritative_dns_workload* =50, *port_id_range*=1, *tau*=5, and *benign_noise_rate*=1000.
The attack probability increases by increasing the number of URL-resolution requests until *number_of_url_requests*=5. From that point on, attack probability remains constant.



For two values of parameter *number_of_url_requests* (*2 and 6*), we vary *authoritative_dns_workload* from 1 to 300; *guess*=50 and *port_id_range*=1, *tau*=5, and *benign_noise_rate*=1000.
Result is coherent with previous graph.

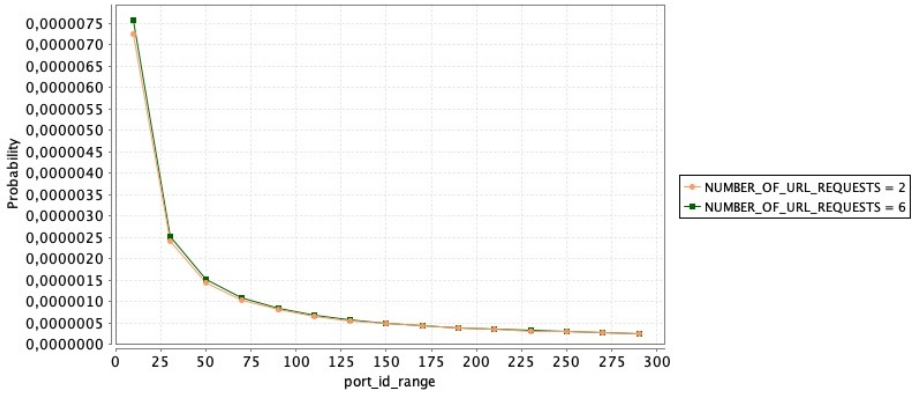
Property verification and experiments

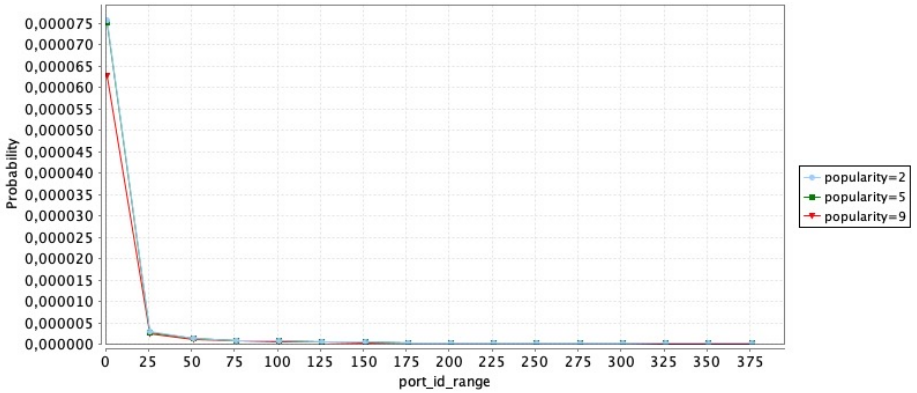


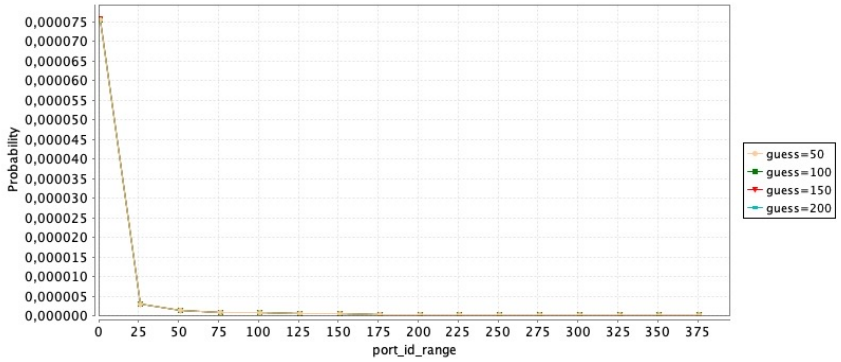
For two values of parameter *number_of_url_requests* (2 and 6), we vary *guess* from 1 to 300; *authoritative_dns_workload* =150, *port_id_range*=1, *tau*=5, *benign_noise_rate*=1000.

Observation: we evince that the only parameter influencing attack success probability is the number of URL requests the attacker implements for the target domain.
In particular, the attack probability changes for $1 \leq number\_of\_url\_requests \leq 5$.

Property verification and experiments







In all the three experiments we can see that as soon as Port randomization is introduced, success probability of the attack goes to zero in a very fast way.

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

Recall DNS Cache-Poisoning attack exploiting fragmentation

Attacker sends a series of 2nd fragment, with different IPIDs, containing the name-IP mapping to poison in resolver's cache → *larger attack window*

Attacker sends a DNS Request to Resolver, which will force Authoritative DNS Server to respond with a fragmented IP packet

No Port number to be guessed

No Query ID to be guessed

Only IPID to be guessed

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

Modules

Intruder Machine
Intruder's PC that requests resolution of a URL

Victim Resolver
Local DNS Server that responds to the client's requests

Root DNS Server
DNS Server that knows the IP address of the Authoritative DNS Server

Domain Server
Authoritative DNS Server of the requested domain

Intruder Server
Intruder's DNS Server trying to guess correct Query ID (and Port number)

- Modules are the *same of S type* attack
- ***Different behaviour*** of *Intruder Server, Victim Resolver*, and *Domain Server*
- ***Different condition*** for *attack success*

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

```
// --- CONSTANTS ---
// Configuration
const int NUMBER_OF_URL_REQUESTS;
const int MAX_QUEUE = NUMBER_OF_URL_REQUESTS + 1;

// DNS Parameters
const double popularity;
const int ipid_range = 65536;

// Rates
const double guess;
const double requests_rate = 1.0;
const double authoritative_dns_workload;

// --- FORMULAS ---
formula cache_poisoned = (correct_guess = true & first_fragment_arrived = true) ; // Attack Status
```


Cache is ***poisoned*** (attack success condition) when attacker ***correctly guesses 2nd fragment*** and the ***legitimate 1st fragment*** from Authoritative DNS Server ***arrives***

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

```
module intruder_machine
        trials : [0..NUMBER_OF_URL_REQUESTS] init 0;
        resolver_replies : [0..NUMBER_OF_URL_REQUESTS] init 0;

        // -- Transitions
        // 1. Send Request: Initiates resolution.
        [DNS_Request_Client_To_Resolver] trials < NUMBER_OF_URL_REQUESTS -> requests_rate : (trials' = trials + 1);

        // 2. Receive Reply: Marks end of query cycle.
        [DNS_Response_Resolver_To_Client] resolver_replies < trials -> (resolver_replies' = resolver_replies + 1);

        // 3. Termination
        [] resolver_replies = NUMBER_OF_URL_REQUESTS -> true;
endmodule
```

Note: all the requests made by the intruder to the resolver imply responses that need to be fragmented

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

```
module intruder_server
        is_state : [0..1] init 0;

        // -- Transitions
        // 1. Start the attack
        [DNS_Request_Client_To_Resolver] is_state = 0 -> (is_state' = 1);

        // 2. Firing Loop
        [Second_Fragment_Guess] is_state = 1 -> guess : (is_state' = 1);

        // 3. Stop the attack
        [DNS_Response_Domain_To_Resolver_Frag2] is_state = 1 -> (is_state' = 0);
endmodule
```

- Attacker ***starts sending 2nd fragment guesses*** *together with the request being sent to the resolver* → We model this behaviour by synchronizing the beginning of the attack with the Intruder Machine sending the request to the resolver.
- Attacker stops if the legitimate 2nd fragment arrives to the resolver.

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

```
module victim_resolver
        ttl : [0..2] init 2; // 0=Miss, 1=Hit, 2=Init
        client_queue : [0..MAX_QUEUE] init 0;
        root_queue : [0..MAX_QUEUE] init 0;
        domain_queue : [0..MAX_QUEUE] init 0;
        responses_queue : [0..MAX_QUEUE] init 0;
        first_fragment_arrived : bool init false;
        second_fragment_arrived : bool init false;

        // -- Variables
        query_to_root_server: bool init false;
        query_to_domain_server: bool init false;
        answer_from_domain_received: bool init false;
        correct_guess: bool init false;

        // -- Transitions
        // 1. Client Request (Merged Hit/Miss Logic)
        [DNS_Request_Client_To_Resolver] (client_queue < MAX_QUEUE) & (responses_queue < MAX_QUEUE) & (root_queue < MAX_QUEUE) ->
                // Branch 1: Cache Hit
                (popularity / 10) : (client_queue' = client_queue + 1) & (ttl' = 1) & (responses_queue' = responses_queue + 1)
                +
                // Branch 2: Cache Miss
                (1 - (popularity / 10)) : (client_queue' = client_queue + 1) & (ttl' = 0) & (root_queue' = root_queue + 1);

        // 2. Query Root
        [DNS_Request_Resolver_To_Root] (client_queue > 0) & (root_queue > 0)
                -> (query_to_root_server' = true);

        // 3. Root Responds -> Move to Domain Queue
        [DNS_Response_Root_To_Resolver] (root_queue > 0) & (domain_queue < MAX_QUEUE)
                -> (root_queue' = root_queue - 1) & (domain_queue' = domain_queue + 1) & (query_to_root_server' = false);

        // 4. Query Domain (Opens Vulnerability Window)
        [DNS_Request_Resolver_To_Domain] domain_queue > 0 -> (query_to_domain_server' = true);

        // 5. Domain Responds with first fragment
        [DNS_Response_Domain_To_Resolver_Frag1] (domain_queue > 0) -> (first_fragment_arrived' = true);
```

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

```
        // 6. Domain Responds with second fragment (Closes Window)
        [DNS_Response_Domain_To_Resolver_Frag2] (responses_queue < MAX_QUEUE) & (domain_queue > 0) & (correct_guess = false)
                -> (domain_queue' = domain_queue - 1) & (responses_queue' = responses_queue + 1) & (query_to_domain_server' = false)
                        & (answer_from_domain_received' = true) & (second_fragment_arrived' = true);

        // 7. Reply to Client
        [DNS_Response_Resolver_To_Client] (responses_queue > 0) & (client_queue > 0) -> (client_queue' = client_queue - 1) & (responses_queue' = responses_queue - 1)
                & (first_fragment_arrived' = false) & (second_fragment_arrived' = false);

        // -- Attack Race Condition
        [Second_Fragment_Guess] (correct_guess = false) & (second_fragment_arrived = false) ->
                1 / (ipid_range) : (correct_guess'=true)
                +
                ((ipid_range - 1) / (ipid_range)) : (correct_guess'=false);

        // -- Sink State
        [] cache_poisoned = true -> true;
endmodule
```

- The attack window closes when resolver receives the legitimate 2nd fragment from Authoritative DNS Server
- The rate at which the attacker correctly guesses the 2nd fragment is given by:

$$guess * \frac{1}{ipid\_range} = guess * \frac{1}{2^{16}}$$

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

```
module root_server
        root_state : [0..1] init 0;

        // -- Transitions
        // 1. Receive Request: A query arrives from the Victim Resolver.
        [DNS_Request_Resolver_To_Root] root_state = 0 -> (root_state' = 1);

        // 2. Send Reply: The Root Server immediately responds, moving the resolver to the next step (Domain Server).
        [DNS_Response_Root_To_Resolver] root_state = 1 -> (root_state' = 0);
endmodule
```

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack

```
module target_domain_server
        // -- States
        domain_state : [0..2] init 0;

        // -- Transitions
        // 1. Receive Request: Opens Race Window
        [DNS_Request_Resolver_To_Domain] domain_state = 0 -> (domain_state' = 1);

        // 2. Legitimate Response: Closes Race Window
        [DNS_Response_Domain_To_Resolver_Frag1] domain_state = 1 -> 1 / (authoritative_dns_workload) : (domain_state' = 2);
        [DNS_Response_Domain_To_Resolver_Frag2] domain_state = 2 -> 1 / (authoritative_dns_workload) : (domain_state' = 0);

endmodule
```

Since DNS response needs to be fragmented, Authoritative DNS Server sends:

1. *1st fragment*
2. *2nd fragment*

Of course, these operations are done together with management of *other legitimate requests arriving to the Authoritative DNS Server*. This gives the attacker higher opportunity to guess the correct 2nd fragment.

Property verification and experiments

P = ? [F cache_poisoned]

Property Verification

Property:
P=? [F cache_poisoned]

Defined constants:
NUMBER_OF_URL_REQUESTS=1,popularity=0,guess=300,authoritative_dns_workload=150

Method:
Verification

Result (probability):
0.6532616691133615 (value in the initial state)

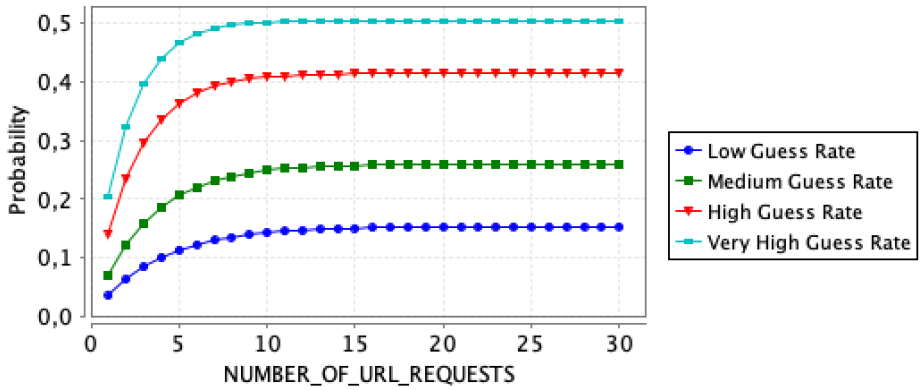
With S_{Frag} *attack*, a high guess rate, and a high workload on the Authoritative DNS Server, the attacker has a ***65%*** of ***probability of poisoning the cache*** of the resolver

This attack has a higher success probability than Kaminsky Attack

Property verification and experiments

P = ? [F cache_poisoned]

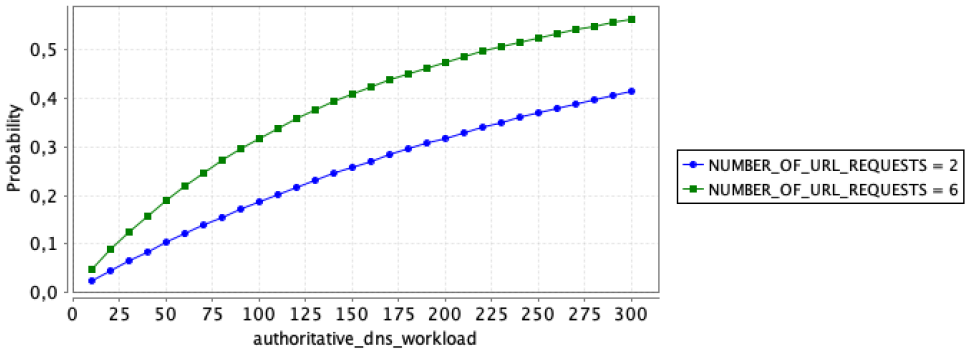
Experiments



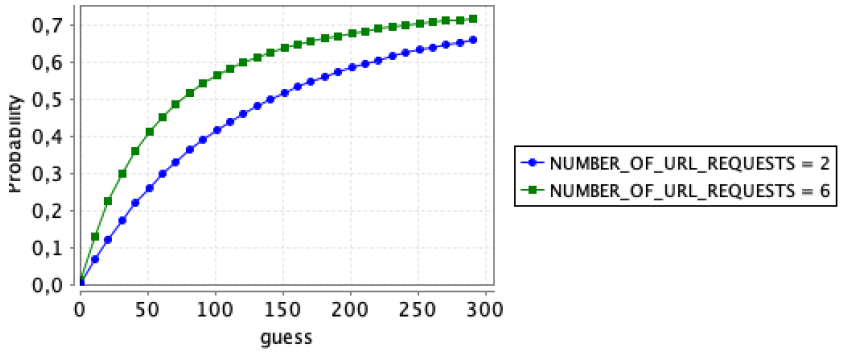
For different values of *guess* rate (low=30, medium=60, high=130, very high=200), *authoritative_dns_workload* =50, and *popularity*=2, we varied *number_of_url_requests* from 1 to 30.
Result shows that probability of attack success increases for *number_of_url_requests* between 1 and 15, and for increasing values of *guess* rate.

Property verification and experiments

P = ? [F cache_poisoned]



For two values of parameter *number_of_url_requests* (*2 and 6*), we vary *authoritative_dns_workload* from 10 to 300; *guess*=50, and popularity=2.
As expected, the higher the workload on Authoritative DNS Server, the higher the probability of attack success.
Moreover, as we've already seen in previous experiment, higher values of *number_of_url_requests* lead to higher probabilities.



For two values of parameter *number_of_url_requests* (2 and 6), we vary *guess* from 1 to 300; *authoritative_dns_workload* =150, and *popularity*=2.
As expected, increasing rate *guess* increases the probability of attack success.

DNS-CPM Mitigation (Rℓ2)

Detection Module (Rℓ2)

↓

Fragments are identified as potential indicators for a DNS Cache-Poisoning Attack.

↓

When the 1st fragment of a response is detected, the response is flagged as a **potential poisoning attack** and passed to the Mitigation Module

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack with DNS-CPM Mitigation

Modules

Intruder Machine
Intruder's PC that requests resolution of a URL

Victim Resolver
Local DNS Server that responds to the client's requests

Root DNS Server
DNS Server that knows the IP address of the Authoritative DNS Server

Domain Server
Authoritative DNS Server of the requested domain

Intruder Server
Intruder's DNS Server trying to guess correct Query ID (and Port number)

DNS-CPM Module
IPS that implements Detection with R∅2 and Mitigation modules

PRISM Model of S_{Frag} DNS Cache-Poisoning Attack with DNS-CPM Mitigation

```
module dns_cpm_module

        // Signal to the Mitigation Module (True if threshold reached)
        detection_signal : bool init false;

        // 1. Detection of a Possible Attack
        [DNS_Response_Domain_To_Resolver_Frag1] true -> (detection_signal' = true);

endmodule
```

Detection module ***kicks in mitigation when a 1st fragment is detected***

```
formula tc_flag_active = (detection_signal = true); // Mitigation Defense
formula cache_poisoned = (correct_guess = true & first_fragment_arrived = true & tc_flag_active = false) ; // Attack Status
```

When mitigation starts, cache cannot be poisoned even if correct 2nd fragment and legitimate 1st fragment has arrived to the resolver

Property verification

P = ? [F cache_poisoned]

Property Verification

Property:
P=? [F cache_poisoned]

Defined constants:
NUMBER_OF_URL_REQUESTS=1,popularity=0,guess=300,authoritative_dns_workload=150

Method:
Verification

Result (probability):
0.0 (value in the initial state)

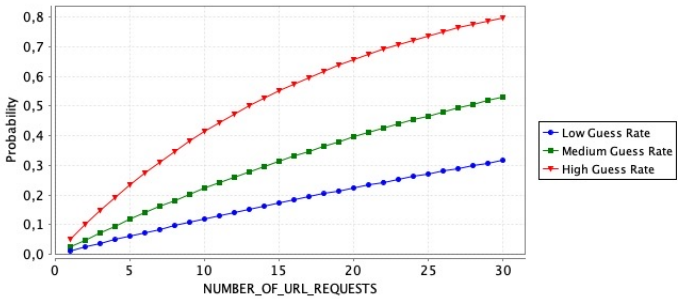
With DNS-CPM R℮2, **0%** of ***probability of poisoning the cache*** of the resolver

This is the result we expected. To poison the cache, both legitimate 1st fragment and correct forged 2nd fragment must arrive to resolver. However, ***as soon as the 1st legitimate fragment arrives***, ***detection module activates mitigation*** which moves conversation to DNS over TCP

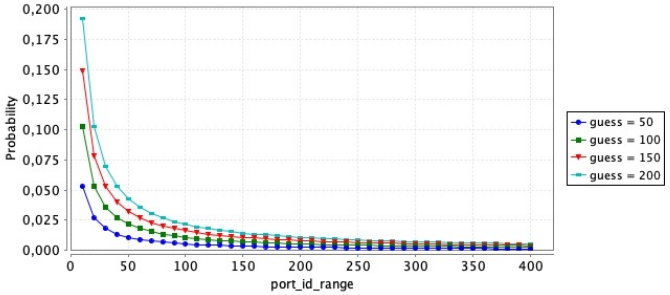
Summarizing

<u>Kaminsky DNS Cache-Poisoning Attack</u>:

- Probability of attack success is strictly related to *guess rate*, *number of URL requests* sent by the attacker to the resolver, and the *workload on* Authoritative DNS Server.
- Attack become more difficult when both Query ID randomization and Port Number randomization are employed.



Randomization of Query ID only



Randomization of Query ID and Port number

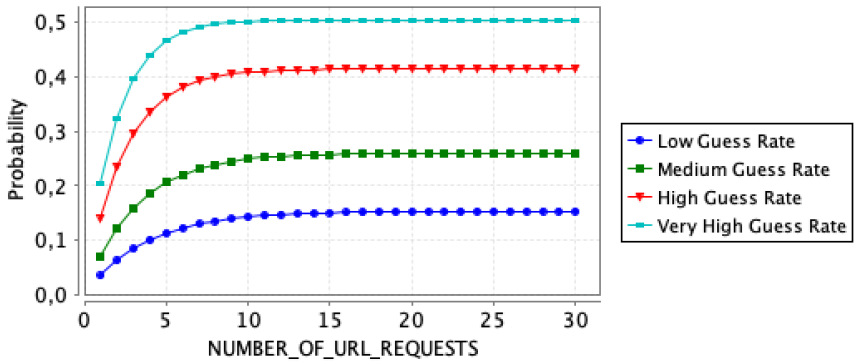
<u>DNS-CPM R*ℓ*1</u>:

- Strongly reduces attack success probability to 0.0076% with only Query ID randomization.
- Notice that once detection triggers the mitigation module, session switches to **DNS-over-TCP**

Summarizing

DNS Cache-Poisoning Attack based on Fragmentation:

- With this type of attack, the intruder can overcome the need of guessing Query ID and Port number.
- It has only to guess a 16-bit IPID.
- To do that, it has a larger window since it starts sending forged 2nd fragment immediately.
- As we've seen, probability of attack is higher than 65%



DNS-CPM Rℓ2:

- Detection with Rℓ2 brings probability of attack success to 0%

Observations and conclusions

Out-of-Bailwick DNS Cache-Poisoning Attack

- Variant of Kaminsky attack
- Approach is the same, but attacker inserts information in Additional Records trying to poison more domain names.
- For this reason, we didn't model it.

DNS-CPM R3:

- It verifies that each record in the DNS response belongs to the same domain (or subdomain) requested in the original query.
- If a record does not belong to the same domain, the packet is immediately classified as suspicious and passed to the mitigation module.
- At that point, session will pass to DNS over TCP, and the attacker cannot poison the resolver's cache.

DNS-CPM Mitigation Limit

Not compatible with the small percentage of resolvers (approx. 2.67%) that ignore the TC flag and do not retry over TCP.

References

Formal Analysis of the Kaminsky DNS Cache-Poisoning Attack Using Probabilistic Model Checking

DNS-CPM: DNS Cache Poisoning Mitigation

Network Security course notes

Computational Models for Complex Systems course notes

PRISM Model Checker

Diagrams Tool: Mermaid JS