2010 IEEE 12th International Symposium on High Assurance Systems Engineering

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

Nikolaos Alexiou, Stylianos Basagiannis
Panagiotis Katsaros

Department of Informatics
Aristotle University of Thessaloniki
Thessaloniki, 54124, Greece
Email: {nalexiou,basags,katsaros}@csd.auth.gr

Tushar Deshpande
Scott A. Smolka

Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400, USA
Email: {tushard,sas}@cs.stonybrook.edu

***Abstract*—We use the probabilistic model checker PRISM to formally model and analyze the highly publicized Kaminsky DNS cache-poisoning attack. DNS (Domain Name System) is an internet-wide, hierarchical naming system used to translate domain names such as google.com into physical IP addresses such as 208.77.188.166. The Kaminsky DNS attack is a recently discovered vulnerability in DNS that allows an intruder to hijack a domain; i.e. corrupt a DNS server so that it replies with the IP address of a malicious web server when asked to resolve URLs within a non-malicious domain such as google.com. A proposed fix for the attack is based on the idea of randomizing the source port a DNS server uses when issuing a query to another server in the DNS hierarchy.**

We use PRISM to introduce a Continuous Time Markov Chain representation of the Kaminsky attack and the proposed fix, and to perform the required probabilistic model checking. Our results, gleaned from more than 240 PRISM runs, formally validate the existence of the Kaminsky cache-poisoning attack even in the presence of an intruder with virtually no knowledge of the victim DNS server's actions. They also serve to quantify the effectiveness of the proposed fix: using nonlinear least-squares curve fitting, we show that the probability of a successful attack obeys a $1/N$ distribution, where N is the upper limit on the range of source-port ids. We also demonstrate an increasing attack probability with an increasing number of attempted attacks or increasing rate at which the intruder guesses the source-port id.

***Keywords*-DNS, Cache Poisoning, Probabilistic Model Checking**

I. Introduction

DNS (Domain Name System) is a hierarchical naming system used to identify network hosts. DNS makes it possible to use a url (Uniform Resource Locator) to address a machine in the internet. It is implemented using DNS name servers, which convert urls into numeric IP addresses. DNS forms the logical backbone of the world wide web, and the service it provides is used on the order of a trillion times a day [4]. Any attack targeting DNS would thus seriously impact the the web's basic operational status, reliability, and security.

In February 2008, security researcher Dan Kaminsky discovered a DNS vulnerability that could be exploited to corrupt normal DNS operation. The attack targets DNS's url-resolution mechanism so that an infected DNS server gives an incorrect IP address for a url. An intruder can exploit this mechanism to *hijack an internet domain*. Specifically, a corrupted DNS server will reply with the IP address of a malicious web server when asked to resolve urls within a non-malicious domain such as google.com. This would direct a large number of unsuspecting clients (ordinary desktop machines) to the malicious web site when they actually wanted to visit a web site within the domain google.com.

In March 2008, some of the world's top DNS experts agreed upon a temporary fix against the attack: randomizing the *source port*, the UDP port a client uses to issue a DNS query. Port randomization [10] means that the intruder must now correctly guess the 16-bit source-port id in addition to the unique 16-bit *query id* assigned to each DNS query. The effective transaction strength thus becomes $2^{16} \cdot 2^{16} = 2^{32}$, as the intruder has to guess a 32-bit number [4].[1]

On August 6, 2008, 30 days after the release of the patch, Kaminsky revealed the nature of vulnerability and how it could be exploited. Thereafter, Kaminsky's attack has received widespread publicity [12], [13]. Note that Kaminsky did not really discover a new attack. Instead, he made clever use of *cache poisoning*, a technique that causes a victimized DNS server to store false information about the IP address associated with a url.

In this paper, we use the probabilistic model checker PRISM [11] to formally analyze the Kaminsky DNS cache-poisoning attack and the effectiveness of the proposed fix. Our approach is to create a Continuous-Time Markov Chain (CTMC) model of the basic DNS url-resolution protocol, the Kaminsky attack, and the proposed fix. A CTMC is a stochastic process that satisfies the Markov property: the conditional probability distribution of future states of the process depend only upon the present state. In a CTMC, the waiting time of

[1] Actually, some of the 2^{16} UDP ports are fixed for certain applications and cannot be used for other purposes [20].

1530-2059/10 $26.00 © 2010 IEEE
DOI 10.1109/HASE.2010.25

Question Section
mail.google.com
Answer Section
209.85.132.83
Authority Section (optional)
Additional Section (optional)

Fig. 1. Authoritative Answer (AA) for a DNS query.

Question Section
mail.google.com
Answer Section (empty)
Authority Section
ns1.google.com
Additional Section
216.239.32.10

Fig. 2. Referral Response (RR) for a DNS query.

a transition from state i to state j is governed by a negative exponential distribution, the parameter of which is *transition rate* q_{ij}. CTMCs are widely used in systems analysis due to their strength in representing dynamic behavior, physical processes, and queueing systems with Poisson arrival rates. They are also amenable to analytical and numerical analysis.

We comprehensively explore our model's multi-dimensional parameter space by systematically varying a number of key parameters, including: the maximum port id, which defines the range of source-port ids, and thus the strength of the proposed fix; the rate at which the intruder launches an attack by sending a corrupted response to the victim DNS server; and the popularity of the target url, which determines how likely it is for the victim DNS server to have a live cache entry for the target url.

Collectively, our results, gleaned from more than 240 runs of the PRISM model checker, formally validate the existence of the Kaminsky DNS cache-poisoning attack even in the presence of an intruder with virtually no knowledge of the victim DNS server's actions. They also serve to quantify the effectiveness of the proposed fix: using nonlinear least-squares curve fitting, we show that the probability of a successful attack obeys a $1/N$ distribution, where N is the maximum source-port id. Additionally, our results demonstrate an increasing attack probability with an increasing number of attempted attacks or increasing rate at which the intruder guesses the source-port id. To the best of our knowledge, we are the first to formally model and analyze the Kaminsky DNS attack and the proposed fix.

The rest of the paper is structured as follows. Section II provides a brief overview of DNS, while Section III describes Kaminsky's attack. Section IV highlights those features of PRISM essential to our analysis. Section V introduces the PRISM model, and Section VI presents our model-checking and curve-fitting results. Section VII considers related work, while Section VIII offers our concluding remarks and directions for future work.

II. DNS

DNS (Domain Name System) is a hierarchical naming system for the internet based on an underlying client-server architecture, which is also hierarchical in nature. The primary function of a DNS server is to perform *url-resolution*: the process of translating a url or domain name, such as mail. google.com, into a physical IP address, such as 209.85.132.83. Domain names and DNS servers are organized hierarchically in terms of top-level domains and subordinate, lower-level domains, respectively com, google and mail in our example.

When a DNS server receives a url-resolution query from a client, typically an ordinary desktop machine, it first checks to see if it can answer the query *authoritatively* based on a locally maintained database of *resource records* mapping domain names to IP addresses. If the queried name matches a corresponding resource record in its local database, the server gives an *authoritative answer* (AA), using the local resource record to resolve the queried name. The structure of an authoritative answer for a DNS query is shown in Fig. 1. The *Question Section* contains the url to be resolved. The *Answer Section* contains the IP address for the url in the Question Section [4]. If no local information exists for the queried name, the server then checks to see if it can resolve the name using information cached locally from previous queries. If a match is found, the server answers with the appropriate cache entry and the query is completed [18].

If the queried name does not find a matched answer at its preferred server—either from its cache or local database—the query process can continue, using *recursion* to fully resolve the name. Such *recursive queries* involve assistance from other DNS servers to help resolve them. The response to the last recursive query is the AA response (if the url is valid). Most DNS servers are configured to support recursive queries, as this is a server's default configuration. An exception to this rule are the so-called root DNS servers for top-level domains, which are configured to be non-recursive. Such a server will instead provide a *referral response* (RR) to a DNS query: a pointer (referral) to another DNS server that presumably has authority for a lower portion of the DNS namespace and can assist in resolving the query. The structure of a referral response for a DNS query is shown in Fig. 2. The *Question Section* contains the url to be resolved. The *Answer Section* is empty. The *Authority Section* and the *Additional Section* respectively contain the name and IP address for the DNS server to which the referral response points [4].

Caching reduces traffic between DNS servers and therefore improves DNS performance. For an AA response, the contents of Answer Section are cached whereas for an RR response, the contents of the Additional Section are cached [4]. To keep cached information from becoming stale and to lessen the demand on authoritative name servers, a server stores DNS query results in its cache for a specific period of time known as

Time To Live (TTL). When a caching (recursive) name server queries an authoritative name server for a resource record, it will cache that record for the time (in seconds) specified by the TTL. If a client queries the caching name server for the same record before the TTL has expired, the caching server will simply reply with the already cached resource record rather than re-retrieve it from the authoritative name server.

III. The Kaminsky DNS Cache-Poisoning Attack

To understand how Kaminsky's DNS attack works, consider the following scenario. A client machine of the authoritative DNS server for the domain cs.sunysb.edu asks this server to resolve a url within the domain google.com. Meanwhile, an *intruder*, who is in control of the domain badguy.com, seeks to poison the cache of this DNS server in such a way that the IP address of badguy.com is substituted for the IP address of DNS server for google.com, the *target domain* of the attack. For reasons that should now be obvious, we refer to this server as the *victim* of the attack and assume it is recursive and therefore caching. In the event of a successful attack, the victim will reply to the client's url-resolution request for a url within the target domain google.com with the IP address of the malicious domain. Here are the exact steps involved in the attack.

1) The intruder configures its own DNS server to be authoritative for the target domain google.com. It then lures a client in the victim's domain to generate DNS lookup queries to resolve a url in the domain controlled by the intruder (badguy.com). The intruder can do this, for example, by claiming to have forgotten a password, prompting the victim to respond by e-mail [13].
2) The victim performs a DNS lookup in order to find out where to send the e-mail.
3) Upon receiving the victim's query, the intruder's name server extracts and saves the port id at which the victim DNS server expects to receive the response (the victim's source port). The intruder's name server also pretends that it is not authoritative for domain badguy.com. It does this by sending the victim an RR response (it should have sent it an AA response), referring the victim to a random name within the target domain, such as 1.google.com, 2.google.com, and 3.google.com. Such a url is unlikely to be in the cache even if other lookups for this domain have been recently performed. Since the intruder now knows that the victim will likely start a DNS lookup for the chosen url, it has an opportunity to attempt to poison the victim's cache [21].
4) On receiving the intruder's response, the victim generates a query to resolve one of the random names within the target domain. The victim assigns a new query id to this request.
5) While the victim is waiting for a legitimate AA response to its query, the intruder tries to provide the victim with a fake RR response. The additional section of a fake RR response contains the IP address of the intruder's DNS server. If the intruder guesses the correct query id, the victim accepts and subsequently caches the fake RR response, thereby corrupting its cache [4]. The intruder can send such messages to the victim since it is configured to be authoritative for the target domain google.com.
6) To increase the likelihood of a successful attack, the intruder floods the victim with many forged packets having different query ids. The intruder needs to do this because the victim assigns a unique query id to each DNS query and only a response packet with a matching query id will be accepted. These forged packets say that the intruder is authoritative for the target domain. As such, upon a successful attack, the intruder will own the entire zone of the target domain [13].

The proposed fix for this attack is to randomize the source port [10]. Rather than use just a single UDP port, which can be easily discovered by the intruder as described above, a much larger range of ports is allocated by a name server and then used randomly when making out-bound queries [4], [17], [8].

IV. Probabilistic Model Checking and PRISM

Probabilistic model checking is the problem of determining the probability by which a probabilistic model M satisfies a probabilistic temporal logic formula φ, where M represent a system model and φ represents a system property. The probabilistic model checker PRISM [6], [11] supports three types of probabilistic models: Markov decision processes (MDPs), discrete-time Markov chains (DTMCs), and continuous-time Markov chains (CTMCs). For the reasons discussed in Section I, we use CTMCs. Properties are specified in PRISM using Probabilistic Computation Tree Logic (PCTL) and, for CTMCs, in an extended version called Continuous Stochastic Logic (CSL). We define properties of the form `F prop`, where `F` is the "eventually" linear temporal operator (sometimes called "Future") and `prop` is a state assertion that evaluates to true or false for a single model state.

A model in PRISM is constructed as the parallel composition of its modules. The behavior of each module is described by a collection of guarded commands, each of which comprises a guard and one or more update actions:

$$\texttt{[]}\ \mathtt{g} \Rightarrow \lambda_1\ \texttt{:}\ \mathtt{u}_1\ \texttt{+}\ \ldots\ \texttt{+}\ \lambda_n\ \texttt{:}\ \mathtt{u}_n\ \texttt{;}$$

The guard `g` is a predicate over model variables. Each update action $\mathtt{u}_i$ describes a transition the module can make by giving the variables new values; in the case of CTMCs, λ_i is the transition's associated rate. If the guard is true, the updates are executed according to their rates.

Commands can be labeled and this provides a mechanism for modules to interact with each other by synchronizing on identically labeled commands. The rate of the resulting transition is the product of the rates of the individual transitions.

V. PRISM Model of the Kaminsky Attack

Based on the 6-step attack scenario described in Section III, we model Kaminsky's DNS attack as a CTMC. Our model is *minimal* in the sense that it contains just enough details

(modules and actions) to reveal the basic vulnerability in DNS that makes Kaminsky's cache-poisoning attack possible.

In modeling the attack, we assume that the intruder has already lured a client of the victim DNS server into generating a query to resolve a url within the domain badguy.com, and that the intruder's DNS server has received the victim's query and now knows the victim's source-port id. These assumptions are valid in the sense that the steps embodied in them are part of the mechanics of launching the attack, and not part of the actual vulnerability that makes the attack possible in the first place. They also serve to simplify our model: model execution can now begin with the intruder launching an attack by having its DNS server send the victim one or more bogus responses, referring it to a number of random urls such as 1.google.com, 2.google.com, and 3.google.com within the target domain.[2] Moreover, these assumptions obviate the need to directly model a client of the victim DNS server.

The architecture of our PRISM CTMC model of the Kaminsky DNS attack, and the actions of the principals involved in the attack, are illustrated in Fig. 3. Our model defines the following four modules, each of which is a DNS server.

- **Client Server (CS):** CS is the victim of the attack. It is recursive, maintains a cache, and is authoritative for the domain cs.sunysb.edu. In order to resolve a url outside of this domain, it contacts the root DNS server; i.e. it has a resource record containing the IP address of the root DNS server. Whenever the victim sends a request to resolve a url, it saves the query id and source-port id of the request in a wait-for-reply queue. If the url is resolved successfully, then its IP address is stored in the url-resolution cache until the TTL expires.
- **Root Server (RS):** RS is the root of the DNS hierarchy. It possesses a resource record containing the IP address of the DNS server for google.com.
- **Domain Server (DS):** DS is the authoritative name server for the target domain (google.com). It sends an AA response to all url-resolution requests seeking to resolve a url within the target domain.
- **Intruder Server (IS):** IS is the authoritative DNS server for the intruder's domain badguy.com.

As discussed in Section III, the proposed fix for the Kaminsky attack is to have a name server using a random 16-bit source port each time it issues a new url-resolution request. Now, the intruder needs to guess the victim's port id in addition to the query id. To model the fix, we introduce the parameter `max_port_id`, which defines the range of source-port ids as `1..max_port_id`. The intruder must now attempt to guess the source-port id, in addition to the query id, from this range. Choosing a value of 1 for `max_port_id` allows us to run the CTMC model with source-port randomization turned off, whereas choosing a value greater than 1 for this parameter allows us to run the model with source-port randomization turned on.

[2]It makes sense for an intruder to attempt to hijack a high-traffic domain such as google.com, as this would presumably impact the greatest number of victim clients.

Further details about the `max_port_id` parameter, along with a description of other key model parameters, is now given. In describing these parameters, we use the term *fake RR response* for the RR messages the intruder sends to the CS declaring itself to be authoritative for the target domain google.com. A *correct guess* (as opposed to an incorrect guess) represents a fake RR response that correctly matches CS's port id. The acceptance of a correct guess by the CS means that the cache-poisoning attack has succeeded.

We also use the term *live cache entry* for a cache entry having a positive TTL (Time To Live) value (TTL > 0). If the CS has a live cache entry for the target domain, it can respond immediately to the intruder's bogus RR response. If the TTL is expired (TTL $= 0$), the CS asks the RS to resolve the target url. The RS cannot resolve the target url but sends the victim an RR response, referring it to the DS. This gives intruder an opportunity to send the victim fake RR responses while the victim awaits a legitimate authoritative response from the DS.

The model parameters are the following:

- **max_port_id:** Defines the range of source port ids as `1..max_port_id` for the purpose of implementing source-port randomization. This is reflected in the model by the rate at which correct guesses arrive at the CS. See the description of parameter `guess` below. As shown in Section VI-A, the attack probability follows a $1/$`max_port_id` distribution. We vary `max_port_id` from 1 to 400, since the probability of a successful cache-poisoning attack is found to be extremely low for `max_port_id` > 400.
- **guess:** The overall rate at which IS sends fake RR responses to CS. These responses may be correct or incorrect guesses, arriving at CS with sub-rates dependent on `guess`. We vary `guess` from 10 to 300 since the probability of a successful cache-poisoning attack remains unchanged for `guess` > 300.
- **popularity:** The rate at which the TTL associated with the CS's cache entry for google.com has a positive value. The more popular the url, the more likely it is to have a live cache entry. Popularity is characterized as low, medium, and high according to its value: a popularity rate of 1-3 is used for less popular sites, 4-7 for medium-popularity sites, and 8-9 for very popular sites.
- **times_to_request_url:** The number of times the IS sends a bogus RR response to the CS, referring it to a random url within the DS and thereby launching a cache-poisoning attack. We vary `times_to_request_url` from 1 to 30.
- **other_legitimate_requests_rate:** The rate at which requests from DNS servers other than CS arrive at the DS. Parameter `other_legitimate_requests_rate` is therefore used to represent the load on the DS. Higher loads mean longer delays for the DS in processing requests and sending back responses. We vary the `other_legitimate_requests_rate` from 1 to 300.

Each module defines certain actions, which synchronize

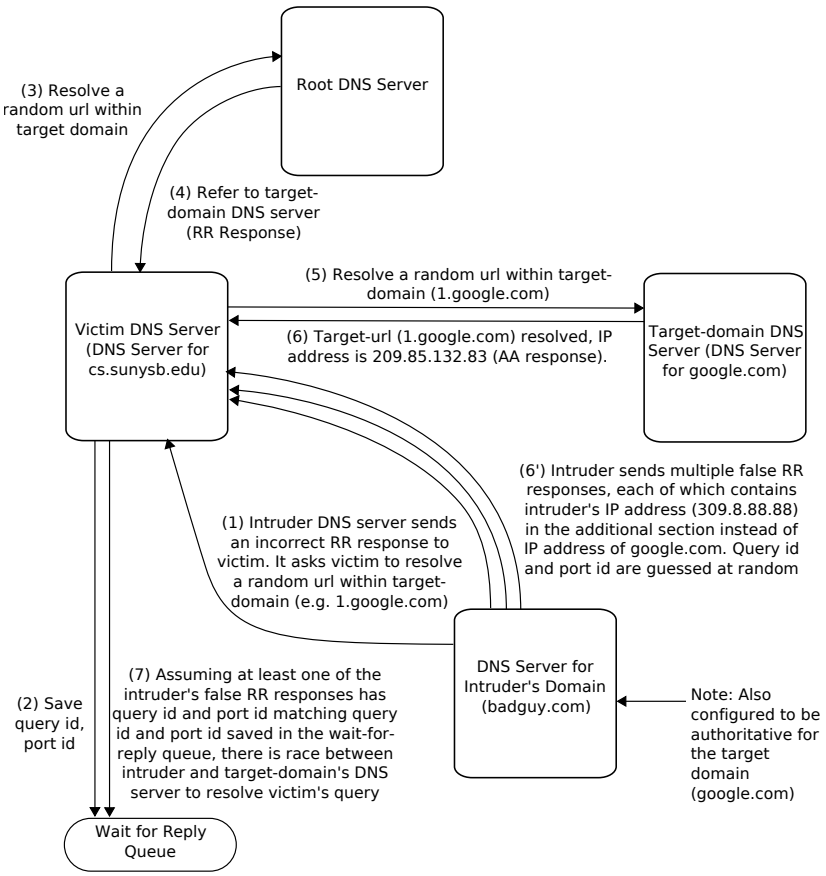

Fig. 3. Architecture of PRISM model of Kaminsky DNS attack.

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

Actions Defined for IS

a) **Send correct guess to CS:** The IS sends a fake RR response to the CS that correctly matches the CS's source-port id, thereby poisoning the cache. This action synchronizes with action <*Receive correct guess from IS*> of CS, and has an associated rate given by parameter `guess`.
b) **Send incorrect guess to CS:** The IS sends a fake RR response to the CS that does not match the CS's source-port id. This action is synchronized with action <*Receive incorrect guess from IS*> of CS, and has an associated rate given by parameter `guess`.

Actions Defined for CS

a) **Send url-resolution request to RS:** With rate $1 - \texttt{popularity}/10$, the TTL of the target url's cache entry is given the value 0. In this case, the requested url does not exist in the cache, and a query is sent to the RS. With rate $\texttt{popularity}/10$, this TTL gets the value 1. In this case, the requested url is cached, and the counter of answered queries is increased by one. This action is synchronized with action <*Process request sent by CS*> of RS.
b) **Receive response from DS:** The response from DS is an authoritative response. In this case, the DS has won the race with the IS and a cache-poisoning attack has been avoided. This action is synchronized with action <*Process request sent by CS*> of DS and action <*Restart attack*> of IS. Its rate is determined by a number of factors, including the rate at which the TTL of the target url is given the value 0.
c) **Receive correct guess from IS:** Let $n = \texttt{max\_query\_id} \cdot \texttt{max\_port\_id}$, where `max_query_id` is the constant 65,536 (2^{16}). This action executes with rate $1/n$ and synchronizes with action <*Send correct guess to CS*> of IS. The combined arrival rate for correct guesses is obtained by multiplying the

rates of the these two synchronizing actions: $(1/n) \cdot$ guess.

d) **Receive incorrect guess from IS:** This action executes with rate $1 - 1/n$ and synchronizes with action *<Send incorrect guess to CS>* of IS. The combined arrival rate for incorrect guesses is obtained by multiplying the rates of the these two synchronizing actions: $(1 - 1/n) \cdot$ guess.

Action Defined for RS

a) **Process request sent by CS:** A url-resolution request is received from the CS. A referral response directing CS to DS is sent to CS. This action is synchronized with the action *<Send url-resolution request to RS>* of CS.

Actions Defined for DS

a) **Process request sent by CS:** A url-resolution request is received from the CS. An authoritative response is sent to CS.

b) **Receive request from other servers:** The DS needs to process requests to resolve target-domain urls from DNS servers other than the victim (CS), thereby increasing its workload and slowing it down. This offers more time for the IS to carry out an attack. Its rate is given by $(1 - 1/$other_legitimate_requests_rate $)$.

CSL Property: We want to determine the *attack probability*, i.e. the probability the intruder carries out a successful attack, which is indicated by the victim having a poisoned url-resolution cache. Therefore, a successful attack arises when the entry in the victim's cache for the target domain (google.com) contains the IP address of IS, the intruder's DNS server. The CSL formula to calculate the attack probability P is therefore: P=? [F cache_poisoned]. The state assertion cache_poisoned becomes true when the IS correctly guesses the victim's source-port id.

VI. Model-Checking and Curve-Fitting Results

In this section, we present six sets of model-checking results (Figs. 4-9) obtained by running PRISM on our CTMC model of the Kaminsky DNS attack. Each result set demonstrates the effect of varying one or more of the five critical model parameters (see Section V) on the attack probability.

Our results are partitioned into two groups. For result sets 1-3, max_port_id $= 1$, meaning that the proposed fix for the Kaminsky cache-poisoning attack is turned off. In this setting, we show the effects on the attack probability of varying parameters times_to_request_url, other_legitimate_requests_rate, and guess, respectively. For result sets 4-6, the effect of source-port randomization on the attack probability is demonstrated by varying max_port_id from 1 to 400. Within this setting, we also

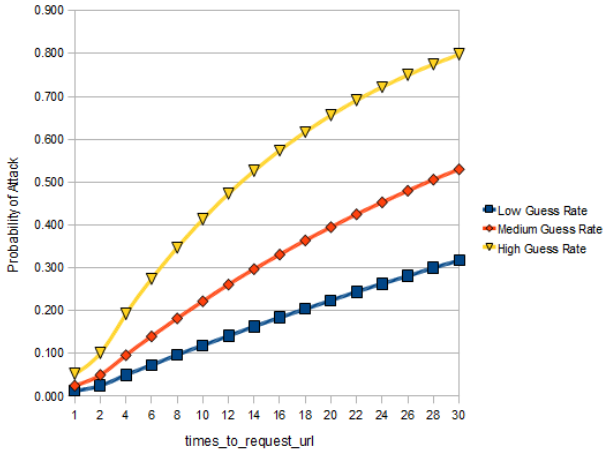

Fig. 4. Results of varying times_to_request_url with max_port_id =1.

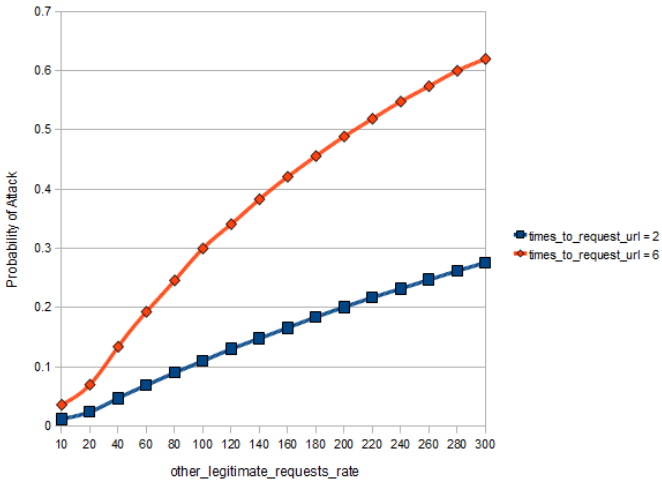

Fig. 5. Results of varying other_legitimate_requests_rate with max_port_id =1.

vary times_to_request_url, guess, and popularity, respectively, demonstrating their second-order effects on the attack probability. For all result sets, the victim's max_query_id is given the fixed value of $2^{16} = 65536$.

Result Set 1: For three different values of parameter guess (low=30, medium=60, high=130), we vary times_to_request_url from 0 to 30; other_legitimate_requests_rate is set to 35 and max_port_id is set to 1.

As Fig. 4 shows, the attack probability increases with increasing times_to_request_url values. This is as expected since the more url-resolution requests there are for the target domain, the more opportunities there are for the IS to carry out a cache-poisoning attack. As a second-order effect, we also observe that by increasing the guess rate, the attack probability is increased: the more opportunities the IS is given to guess the source-port id, the greater its probability of doing so.

Result Set 2: For two different values of times_to_request_url (2 and 6), we vary other_legitimate_re-

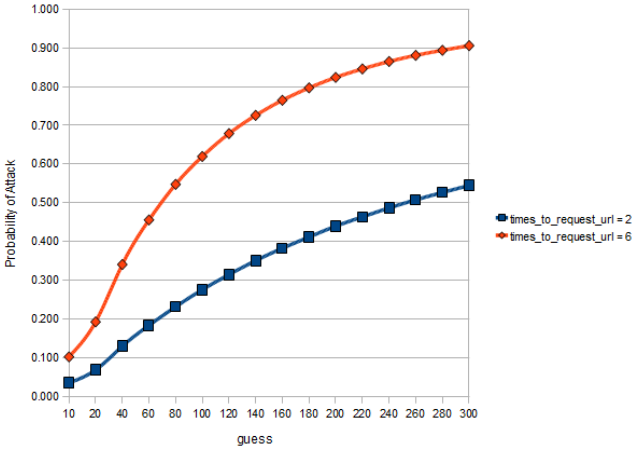

Fig. 6. Results of varying guess with max_port_id =1.

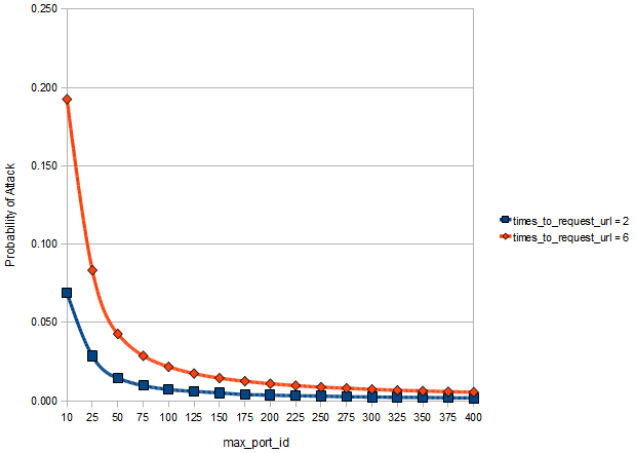

Fig. 7. Results for different times_to_request_url values while varying max_port_id.

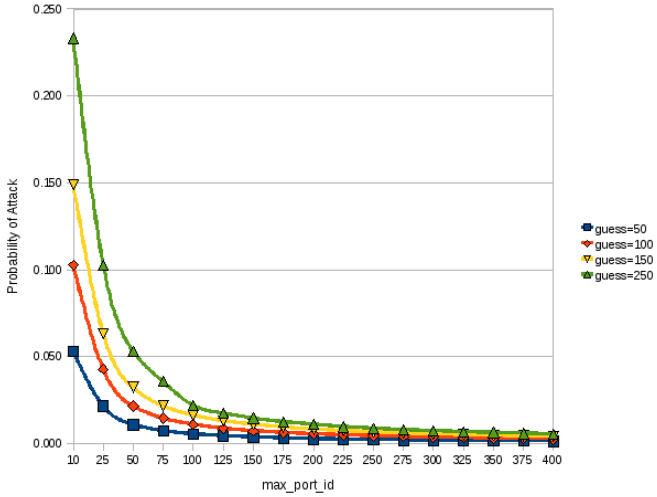

Fig. 8. Results for different guess rates while varying max_port_id.

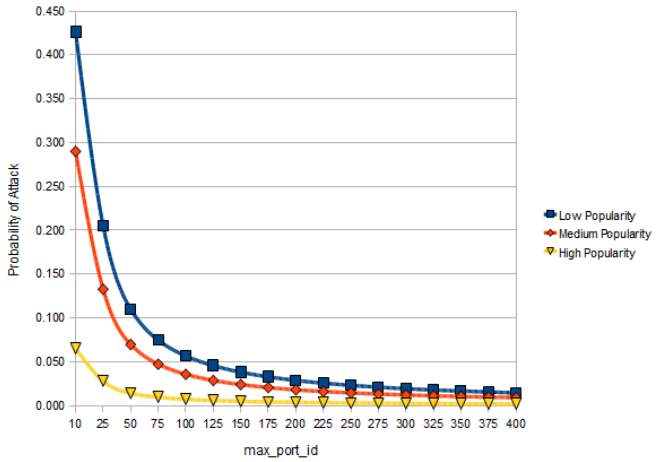

Fig. 9. Results for different popularity values while varying max_port_id.

quests_rate from 10 to 300; the guess rate is set to 50 and max_port_id is set to 1.

As Fig. 5 shows, the attack probability increases with increasing other_legitimate_requests_rate values. As other_legitimate_requests_rate increases, so does the workload on DS, resulting in increasingly longer delays in responding to CS queries. Moreover, recall that the IS is in a race with the DS to respond to a CS query, and should it win the race, cache-poisoning ensues. Therefore, the longer the DS is delayed processing other url-resolution requests, the greater the probability of cache poisoning. Also, as explained above, the attack probability is higher for a higher value of times_to_request_url.

Result Set 3: For two different values of times_to_request_url (2 and 6), we vary rate guess from 10 to 300; other_legitimate_requests_rate is set to 150 and max_port_id is set to 1.

Fig. 6 demonstrates the impact of increasing the guess rate on the attack probability. With source-port randomization turned off (max_port_id =1), to poison the cache, the IS need only correctly guess the query id and for this correct guess to reach the CS ahead of the DS's AA reply. Increasing rate guess increases the probability of this happening. The second-order effects of increasing times_to_request_url are also demonstrated.

Result Sets 4-6: For each of these result sets, we vary max_port_id from 1 to 400. For result set 4, we additionally consider two different values of times_to_request_url (2 and 6), while setting guess to 200 and other_legitimate_requests_rate to 150. For result set 5, we consider four different values of guess (50,100, 150, and 200), while setting times_to_request_url to 6 and other_legitimate_requests_rate to 150. For result set 6, we consider three different values of popularity (low=2, medium=5, high=9), while setting guess to 200, times_to_request_url to 8, and other_legitimate_requests_rate to 300. Result sets 4 and 5 were obtained with a popularity value of 2 (low).

As Figs. 7-9 show, the probability of a successful attack de-

Fig.	guess	olrr	pop.	ttru	A	B
7	200	150	2	2	0.692	0.0002
	200	150	2	6	1.934	0.0018
8	50	150	2	6	0.531	0.0002
	100	150	2	6	1.033	0.0004
	150	150	2	6	1.493	0.0009
	250	150	2	6	2.369	0.0002
9	200	300	2	8	4.290	0.0097
	200	300	5	8	2.917	0.0046
	200	300	9	8	0.652	0.0005

TABLE I
CURVE-FIT ANALYSIS OF FIGS. 7-9.

creases nonlinearly as the value of max_port_id increases, due to the fact that, with source-port randomization turned on, the intruder needs to guess the correct source-port id from a much larger range. We in fact show in Section VI-A that each of these plots follows the $1/N$ distribution, where $N =$ max_port_id.

The results of Figs. 7-9 also serve to demonstrate the second-order effects of varying parameters times_to_request_url, guess, and popularity, respectively. The impact of parameter settings for times_to_request_url and guess have already been considered above. In the case of popularity, the lower the value, the greater the probability the requested url is *not* cached at the victim CS. Should this be the case, the CS will initiate a recursive query to resolve the target url, giving the IS an opportunity to carry out a cache-poisoning attack. This explains why a a lower popularity value results in uniformly higher attack probabilities for all possible max_port_id values.

A. Curve-Fitting Analysis of Model-Checking Results

From Figs. 7-9, we observed that with source-port randomization turned on, the probability of a successful attack decreases nonlinearly as the value of max_port_id increases. To determine the precise nature of this effect, we subjected the results of Figs. 7-9 to nonlinear least-squares curve fitting using Gnuplot [7]. Gnuplot minimizes the weighted least-squares (chi-square) merit function. We used a convergence criterion of 10^{-5}. Fig. 10 shows the results of our curve-fit analysis for the three datasets plotted in Fig. 9. Table I summarizes our curve-fitting analysis of the nine total plots of Figs. 7-9, and indicates the parameter settings used for each dataset. (In the table, column headings ttru, olrr and pop are abbreviations for times_to_request_url, other_legitimate_requests_rate and popularity respectively.) Columns A and B are explained below.

Our curve-fitting results of Fig. 10 and Table I reveal that the attack probability distribution is of the form $A/x + B$, where $x =$ max_port_id, A ranges from 0.531 to 4.290, with an average value of 1.768, and the values of parameter B are very small (on the order of 10^{-3}) and thus can be ignored. We may therefore conclude that *the attack probability for our model is inversely proportional to* max_port_id.

B. Validation of Results and Runtime Statistics

We used the PRISM simulator to confirm the existence of both winning and losing intruder execution sequences. A winning execution results in the poisoning of the victim's cache, while a losing one implies that the target domain (google.com) is resolved correctly. In doing so, we observed that there are two kinds of winning execution sequences for the intruder:

1) Response from intruder arrives at victim before referral response from RS reaches victim.
2) Response from intruder arrives at victim before authoritative response from DS reaches victim.

The intruder is therefore in a race with the RS as well as DS, and the attempted attack is successful if it wins either race.

We also observed that for optimal attack settings max_query_id = 1 (least), max_port_id = 1 (least), times_to_request_url = 30 (high), guess = 300 (high), other_legitimate_requests_rate = 300 (high), and popularity = 2 (low), the attack probability is 0.999, which is almost 1. This result is as expected and further serves to validate the model.

Tables II-IV contain various statistics for our CTMC model corresponding to the medium-guess-rate curve of Fig. 4. (In the tables, column heading ttru is an abbreviation for times_to_request_url.) More specifically, we executed our model with guess = 60, other_legitimate_requests_rate = 35, max_port_id = 1 and times_to_request_url ranging from 1 to 30. Table II provides basic statistics for both the CTMC model and the MTBDD PRISM uses to represent the model's reachable state space. For the CTMC, the number of states and number of non-zero transitions are given; for the MTBDD, the total number of nodes and the amount of memory needed to store the MTBDD are given [3]. A good estimate for the size of each node is 20 bytes. The memory usage is thus given by the formula $Memory(KB) = total\ number\ of\ nodes \cdot 20/1024$. For all executions, the MTBDD had a single initial state and 8 terminal nodes (leaves).

Table III shows the times taken to construct the model, a two-step process. In the first step, a CTMC (represented as an MTBDD) is created from the system description. In the second step, the reachable states are computed using a BDD-

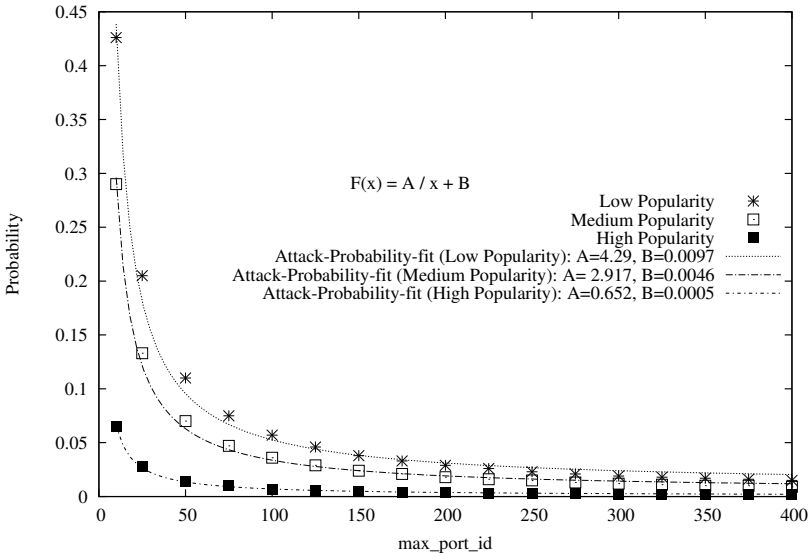

Fig. 10. Attack probability and curve fit for results from Fig. 9

ttru	Model		MTBDD	
	States	Transitions	Nodes	Memory (KB)
1	10	13	352	6.88
5	1232	4272	6963	136.00
10	14992	65682	20127	393.11
15	67777	67777	37990	741.99
20	201087	996727	52314	1021.76
25	471422	2397362	66651	1301.78
30	950282	4917072	80886	1579.80

TABLE II
GENERAL STATISTICS FOR PRISM CTMC MODEL

ttru	No. Iterations	Model Construction Time (sec)
1	7	0.004
5	21	0.090
10	36	0.670
15	51	2.360
20	66	5.290
25	81	14.290
30	96	22.930

TABLE III
CONSTRUCTION-TIME STATISTICS FOR PRISM CTMC MODEL

based fixpoint algorithm [3]. The number of fixpoint iterations and the time required for them is given in Table III.

Table IV gives the times taken to compute the attack probability using the Jacobi Over-relaxation (JOR) method. JOR is the standard method used by PRISM's MTBDD engine [3]. The table gives the number of iterations performed during model checking, the time taken for model checking, and the attack probabilities.

All results were obtained on an Intel Core 2 Duo Processor with 4 GB RAM and dual 1.66 GHz Intel Centrino T5500 processors, each with 2 MB L2 cache. The OS was

ttru	No. Iterations	Model Checking Time (sec)	Attack Probability
1	6	0.010	0.025
5	30	0.040	0.118
10	60	0.370	0.222
15	90	1.620	0.314
20	120	6.160	0.395
25	150	16.190	0.467
30	180	31.710	0.530

TABLE IV
MODEL CHECKING STATISTICS FOR PRISM CTMC MODEL

Ubuntu 8.04. Tables II-IV exhibit a significant increase in the size of the model and corresponding model-construction and model-checking times with an increase in the range of `times_to_request_url`.

The size of the state space explored by PRISM while executing the model depends on the values of the parameters that are used to define *pre-conditions* for the various actions. In our PRISM model of the Kaminsky DNS attack, parameter `times_to_request_url` alone appears in pre-conditions. All other parameters are used to define *rates* associated with various actions. Parameter `times_to_request_url` is central to the model in that it defines the initial state of the model's state space. When model execution begins, the IS generates a request to resolve the target url. More requests result in the generation of more url-resolution requests, which in turn cause exploration of bigger and bigger state spaces. So, clearly, the size of the model's state space increases with the value of `times_to_request_url`, leading to an increases in the size of the model and model-execution time.

VII. RELATED WORK

In related work, a number of researchers have deployed probabilistic model checking to analyze threat levels in computer systems and security protocols [9], [16], [14], [2].

Probably the most closely related work is that of [2], where PRISM is used to systematically quantify DoS (Denial of Service) security threats. In [1], Hidden Markov Models are used to develop anomaly-based intrusion detection systems. In [15], generation and analysis of a system's attack graph is used to decide which security vulnerabilities would be most cost-effective to guard against. To the best of our knowledge, we are the first to formally model and analyze the highly publicized Kaminsky DNS attack.

VIII. Conclusions

We have used the PRISM probabilistic model checker to formally model and analyze the highly publicized Kaminsky DNS cache-poisoning attack. The nature of the proposed fix—randomizing a DNS server's source port—made the Kaminsky DNS attack an ideal candidate for probabilistic model checking. Moreover, since the Kaminsky attack is aimed at DNS servers, it was at once both natural and beneficial to model the attack in PRISM as a CTMC, with corresponding arrival rates for benign and malicious requests and their responses.

The results we obtained from this CTMC model formally validate the existence of the attack even in the presence of an intruder with virtually no knowledge of the victim DNS server's actions. They also serve to quantify the effectiveness of source-port randomization as a counter-measure. In particular, our curve-fitting analysis shows that the attack probability is inversely proportional to parameter `max_port_id`, which determines the range of source-port randomization. Our results further demonstrate an increasing probability of successful attack with an increasing number of attempted attacks (`times_to_request_url`), increasing load on the target domain server (`other_legitimate_requests_rate`), and increasing rate of intruder guesses (`guess`). Furthermore, assigning a lower popularity value to the target url also resulted in a higher attack probability.

Port randomization is a short-term fix for the DNS vulnerability Kaminsky discovered. A potential long-term fix is DNSSEC, which tries to prevent cache-poisoning attacks by allowing Web sites to verify their domain names and corresponding IP addresses using digital signatures and public-key encryption [5]. As future work, we plan to extend our PRISM-based analysis to DNSSEC as well as threats that may have been overlooked by DNSSEC, including DNS bandwidth amplification attacks [19]. We believe that probabilistic model checking and in particular CTMC analysis, can be used to quantitatively evaluate and compare security threats, and in the process provide valuable feedback for the design of appropriate countermeasures.

The source files for our PRISM model of the Kaminsky DNS attack, along with all result sets and corresponding settings for model parameters and PRISM command-line options, are available from http://www.cs.sunysb.edu/~sas/kaminsky/.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments. Research supported in part by NSF Grants CCF-0926190 and CCF-1018459 and AFOSR Grant FA0550-09-1-0481. Part of Professor Katsaros's research was conducted while on Sabbatical leave at Stony Brook University.

References

[1] D. Ariu, G. Giacinto, and R. Perdisci. Sensing attacks in computers networks with Hidden Markov Models. In P. Perner, editor, *Proc. 5th International Conference on Machine Learning and Data Mining in Pattern Recognition (MLDM '07)*, volume 4571 of *LNAI*, pages 449–463. Springer, 2007.

[2] S. Basagiannis, P. Katsaros, and A. Pombortsis. Probabilistic model checking for the quantification of DoS security threats. *Computers & Security*, 28(6):450–465, September 2009.

[3] T. Ciardo. Kanban manufacturing system (http://www.prismmodelchecker.org/casestudies/kanban.php).

[4] S. Friedl. An illustrated guide to the Kaminsky DNS vulnerability. *Unixwiz.net Tech Tips*, August 2008 (http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html).

[5] D. Gordon and I. Haddad. *The Basics of DNSSEC*. O'Reilly Media, 2004.

[6] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.

[7] P. Janert. *Gnuplot In Action*. Manning Publications, 2009.

[8] Y. Kadakia. Dan Kaminsky's DNS cache poisoning vulnerability explained. *Yash Kadakia's Blog : One Perspective on Indian IT Security*, August 2008 (http://www.yashkadakia.com/2008/08/dam-kaminskys-dns-cache-poisoning.html).

[9] R. Lanotte, A. Maggiolo-Schettini, and A. Troina. Automatic analysis of a non-repudiation protocol. In *Proc. 2nd International Workshop on Quantitative Aspects of Programming Languages (QAPL'04)*, 2004.

[10] M. Larsen. Port randomization. *IETF Internet Draft*, July 2009 (http://tools.ietf.org/html/draft-ietf-tsvwg-port-randomization-04).

[11] Z. Ma and M. Kwiatkowska. Modelling with PRISM of intelligent system. Master's thesis, Linacre College, University of Oxford, September 2008 (http://www.prismmodelchecker.org/papers/zhongdanma-mscthesis.pdf).

[12] J. Markoff. Leaks in patch for web security hole. *The New York Times*, August 2008.

[13] E. Naone. The flaw at the heart of the internet. *Technology Review*, November/December 2008 (https://www.technologyreview.com/web/21537/).

[14] G. Norman and V. Shmatikov. Analysis of probabilistic contract signing. *Journal of Computer Security*, 14(6):561–589, 2006.

[15] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proc. 2002 IEEE Symposium on Security and Privacy*, 2002.

[16] V. Shmatikov. Probabilistic model checking of an anonymity system. *Journal of Computer Security*, 12:355–377, 2004.

[17] US CERT (United States Computer Emergency Response Team). Vulnerability Note VU#800113 : Multiple DNS implementations vulnerable to cache poisoning. Technical report, US CERT Vulnerability Notes Database, July 2008 (http://www.kb.cert.org/vuls/id/800113).

[18] Microsoft TechNet. How DNS query works. January 2005 (http://technet.microsoft.com/en-us/library/cc775637(WS.10).aspx).

[19] R. Vaughn and G. Evron. DNS amplification attacks. March 2006 (http://www.isotf.org/news/DNS-Amplification-Attacks.pdf).

[20] List of TCP and UDP port numbers (http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers).

[21] C. Wright. Understanding Kaminsky's DNS bug. *Cory Wright's blog*, July 2008 (http://www.linuxjournal.com/content/understanding-kaminskys-dns-bug).