Stony Brook University

Model Checking the Kaminsky DNS Cache-Poisoning Attack Using PRISM

A Thesis Presented

by

Tushar Suhas Deshpande

to

The Graduate School

in partial fulfillment of the requirements for the degree

of

Master of Science

in

Computer Science

Stony Brook University

May 2010

Stony Brook University

The Graduate School

Tushar Suhas Deshpande

We, the thesis committee for the above candidate for the

Master of Science degree, hereby recommend

acceptance of this thesis.

Dr. Scott A. Smolka—Thesis Adviser
Professor, Department of Computer Science

Dr. Scott D. Stoller—Thesis Committee Chair
Professor, Department of Computer Science

Dr. Erez Zadok
Associate Professor, Department of Computer Science

This thesis is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Thesis

Model Checking the Kaminsky DNS Cache-Poisoning Attack Using PRISM

by

Tushar Suhas Deshpande

Master of Science

in

Computer Science

Stony Brook University
2010

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 like `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 the URL of 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 requisite 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, demonstrating an exponentially decreasing, long-tail trajectory for the probability of a successful attack with an increasing range of source-port ids, as well as an increasing attack probability with an increasing number of attempted attacks or increasing rate at which the intruder guesses the source-port id.

To my parents

Contents

List of Figures vi

List of Tables vii

1 Introduction 1

2 DNS 3

3 The Kaminsky DNS Cache-Poisoning Attack 5

4 Probabilistic Model Checking and PRISM 7

5 PRISM Model of the Kaminsky Attack 8
- 5.1 Architecture . . . 8
- 5.2 Parameters . . . 10
- 5.3 Actions . . . 10
- 5.4 CSL Property . . . 12

6 Experimental Results 13
- 6.1 Result Set 1 . . . 14
- 6.2 Result Set 2 . . . 15
- 6.3 Result Set 3 . . . 15
- 6.4 Result Sets 4-6 . . . 15
- 6.5 Validation of Results and Runtime Statistics . . . 16

7 Related Work 20

8 Conclusions 21

Bibliography 22

List of Figures

5.1 Architecture of PRISM model of Kaminsky DNS attack. 9

6.1 Results of varying `times_to_request_url` with `max_port_id` =1 . 13
6.2 Results of varying `other_legitimate_requests_rate` with `max_port_id` =1 14
6.3 Results of varying `guess` with `max_port_id` =1 14
6.4 Results for different `times_to_request_url` values while varying `max_port_id` values 15
6.5 Results for different `guess` rates while varying `max_port_id` values . . 16
6.6 Results for different `popularity` values while varying `max_port_id` values 17

List of Tables

6.1 General Statistics for PRISM CTMC Model 18
6.2 General Statistics for PRISM CTMC Model 18
6.3 Model Checking Statistics for PRISM CTMC Model 18

Acknowledgments

This thesis is based on the research paper titled “Model Checking the Kaminsky DNS Cache-Poisoning Attack Using PRISM” and co-authored with Nikolaos Alexiou, Stylianos Basagiannis and Panagiotis Katsaros from Department of Informatics, Aristotle University of Thessaloniki and Scott A. Smolka from Department of Computer Science, Stony Brook University. This paper has been submitted to the *European Symposium on Research in Computer Security (ESORICS 2010)*.

My immense gratitude to my advisor, Dr. Scott A. Smolka, for his constant guidance and motivation. I thank Dr. Scott D. Stoller and Dr. Erez Zadok for being on my Masters thesis committee and providing valuable suggestions.

Chapter 1

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 [6]. Therefore, any attack targeting DNS would have a serious impact on 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 a domain. Specifically, a corrupted DNS server will reply with the IP address of a malicious web server when asked to resolve the url for 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 `google.com`.

In March 2008, some of the world's top DNS experts agreed upon a temporary but effective fix against the attack: randomizing the *source port*, the UDP port a client uses to issue a DNS query. Port randomization [11] 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, thereby rendering the attack computationally infeasible [6].

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 [13, 15]. Note that Kaminsky had not really discovered 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 show how the probabilistic model checker PRISM [12] can be used to formally model the Kaminsky DNS cache-poisoning attack, in order to analyze the attack dynamics and the effectiveness of the proposed fix. The nature of the fix, which is based on the idea of randomizing the source port associated with a DNS query, makes the attack an ideal candidate for probabilistic model checking. Our approach is to create a multi-dimensional CTMC (Continuous-Time Markov Chain) model of the basic DNS

url-resolution protocol, the Kaminsky attack, and the proposed fix. In a CTMC model, the waiting time of a transition from state i to state j is governed by a negative exponential distribution, the parameter of which is *transition rate* q_{ij}.

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 the victim DNS server is 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, demonstrating an exponentially decreasing, long-tail trajectory for the probability of a successful attack with an increasing range of source-port ids, as well as 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 2 provides a brief overview of DNS, while Section 3 describes Kaminsky's attack. Section 4 highlights those features of PRISM essential to our analysis of the Kaminsky DNS attack. Section 5 presents our PRISM model of the Kaminsky attack, while Section 6 contains our experimental results. Section 7 considers related work, while Section 8 offers our concluding remarks and directions for future work.

Chapter 2

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 `google.com`, into a physical IP address, such as `208.77.188.166`. Domain names and DNS servers are organized hierarchically in terms of top-level domains and subordinate, lower-level domains, respectively `com` and `google` 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. 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 [21].

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. Most DNS servers are configured to support recursive queries, as this is a server's default configuration. An exception to the 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.

Caching reduces traffic between DNS servers and therefore improves DNS performance. 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.

Chapter 3

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 the url `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 `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 the `google.com` with the IP address of the malicious domain. Here are the exact steps involved in the attack.

1. The intruder 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 [15].
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 another server, namely, that of the target domain (`google.com`). Since the intruder now knows that the victim will start a DNS lookup for that server, the intruder has an opportunity to attempt to poison the victim's cache.
4. On receiving the intruder's response, the victim generates a query to resolve the target domain. The victim assigns a new query id to this request.
5. While the victim is waiting for the reply to its query, the intruder tries to supply a false response before the legitimate server can supply the legitimate response. If the intruder guesses the correct query id, the victim accepts the false response, poisoning its own cache.

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 [15].

The proposed fix for this attack is to randomize the source port [11]. 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 [6, 20, 9].

Chapter 4

Probabilistic Model Checking and PRISM

Probabilistic model checking is the problem of given a probabilistic model M and a formula φ of a probabilistic temporal logic, determine the probability by which M satisfies φ. The probabilistic model checker PRISM [8, 12] supports three types of probabilistic models: Markov decision processes (MDPs), discrete-time Markov chains (DTMCs), and continuous-time Markov chains (CTMCs). For the present study, 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:

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

The guard `g` is a predicate over model variables. Each update action $\texttt{u}_i$ 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. In our CTMC model of the Kaminsky DNS attack, each guarded command comprises a single update action ($n = 1$). We therefore subsequently use the terms command and action interchangeably.

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.

Chapter 5

PRISM Model of the Kaminsky Attack

Based on the 6-step attack scenario described in Section 3, 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 a bogus response, referring it to the target domain `google.com`.[1] Moreover, these assumptions obviate the need to directly model a client of the victim DNS server.

5.1 Architecture

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. 5.1. 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

[1] 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's clients.

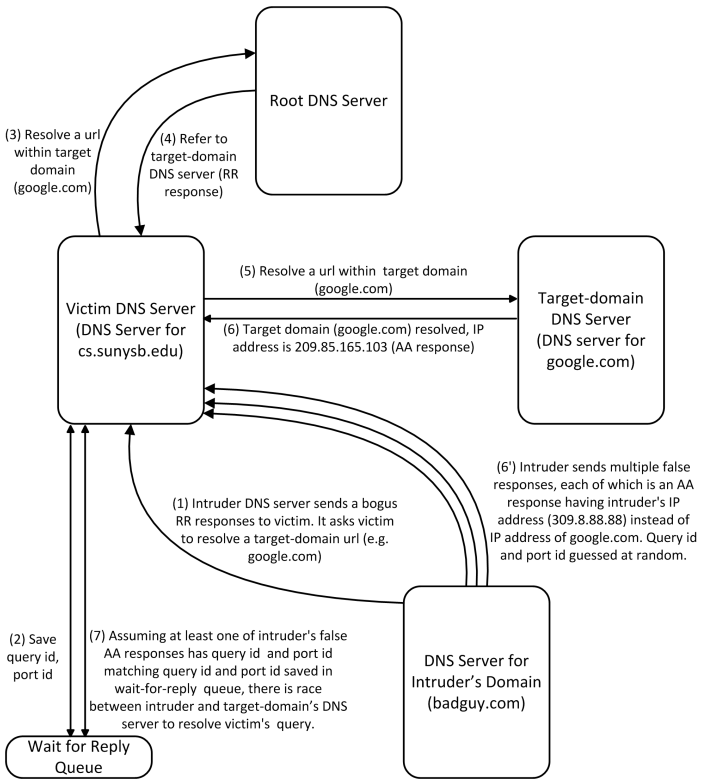

Figure 5.1: Architecture of PRISM model of Kaminsky DNS attack.

(`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 3, 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.

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 AA response* for the AA messages the intruder sends to the CS falsely claiming to be authoritative for the target domain `google.com`. A *correct guess* (as opposed to an incorrect guess) represents a fake AA 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. The intruder now has the opportunity to send the victim fake AA responses while the victim awaits a legitimate authoritative response from the DS.

5.2 Parameters

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: `1/(max_query_id · max_port_id)`, where `max_query_id` is the constant 65,536 (2^{16}). As shown in Section 6, the attack probability follows an exponentially decreasing trajectory with increasing `max_port_id` values. We vary `max_port_id` from 1 to 400, since the probability of a successful cache-poisoning attack is found to be very low for `max_port_id` $>$ 400.
- **guess:** The rate at which IS sends fake AA responses to the CS. These responses may be correct or incorrect guesses. 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 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 `other_legitimate_requests_rate` from 1 to 300.

5.3 Actions

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

Actions Defined for IS

a. **Refer CS to DS:** In response to a CS query to resolve a url within the IS's domain, the IS pretends that it is not authoritative for its own domain (`badguy.com`). It does this by sending a bogus RR response to the CS referring it to the DS. This action is executed if *number of remaining trials* $<$ `times_to_re-`

quest_url; has a constant rate of 1.0, meaning that the IS sends one RR response to the CS per unit time; and is synchronized with action <*Receive bogus RR response from IS*> of CS.

b. **Prepare to carry out attack:** This action is synchronized with action <*Send url-resolution request to DS*> of CS and action <*Receive request from CS*> of DS. Its purpose is to notify the IS that the race between the IS and the DS has begun.
c. **Send correct guess to CS:** The IS sends a fake AA 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 that depends on the parameter guess.
d. **Send incorrect guess to CS:** The IS sends a fake AA 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 that depends on the parameter guess.
e. **Restart attack:** This action is synchronized with action <*Reply to CS*> of DS and action <*Receive response from DS*> of CS. Its purpose is to notify the IS that it has lost the race with the DS and it should now initiate another cache-poisoning attack.

Actions Defined for CS

a. **Receive bogus RR response from IS:** This action is synchronized with action <*Refer CS to DS*> of IS. With rate popularity/10, the TTL of the target url's cache entry gets the value 1. In this case, the requested url is cached, and the counter of answered queries is increased by one.
b. **Send url-resolution request to RS:** With rate $1 -$ popularity/10, the TTL 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. This action is synchronized with action <*Receive request from CS*> of RS.
c. **Receive response from RS:** The response from RS is a referral response. This action is synchronized with the action <*Reply to CS*> of RS.
d. **Send url-resolution request to DS:** A url-resolution query is sent to the DS. This action is synchronized with action <*Receive request from CS*> of DS and action <*Prepare to carry out attack*> of IS.
e. **Receive response from DS:** The response from DS is an authoritative response. A counter is incremented to indicate that a response has been received for a pending query. 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 <*Reply to 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.
f. **Receive correct guess from IS:** Correct guesses by the IS arrive at CS with rate 1/(query_id · max_port_id). This action 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/max_query_id · max_port_id) · guess`.

g. **Receive incorrect guess from IS:** Incorrect guesses by the IS arrive at CS with rate `query_id · max_port_id − 1`. This action synchronizes with action <*Send incorrect guess to CS*> of IS. The combined arrival rate for incorrect guesses is therefore `(max_query_id · max_port_id − 1) · guess`.

Action Defined for RS

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

b. **Reply to CS:** The referral response prepared in conjunction with action <*Receive request from CS*> is sent to the CS. This action is synchronized with action <*Process response received from RS*> of RS.

Actions Defined for DS

a. **Receive request from CS:** A url-resolution request is received from the CS. An authoritative response is prepared. This action is synchronized with actions <*Send url-resolution request to DS*> of CS and <*Prepare to carry out attack*> of IS.

b. **Reply to CS:** The authoritative response prepared in conjunction with action <*Receive request from CS*> is sent to the CS. This action is synchronized with actions <*Receive response from DS*> of CS and <*Restart attack*> of IS. The reply rate is related to the DS's workload and is given by `1/(other_legitimate_requests_rate)`.

c. **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 `(other_legitimate_requests_rate-1)/other_legitimate_requests_rate`.

5.4 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 url (`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.

Chapter 6

Experimental Results

In this section, we present six sets of results (Figs. 6.1-6.6) 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 5) on the attack probability.

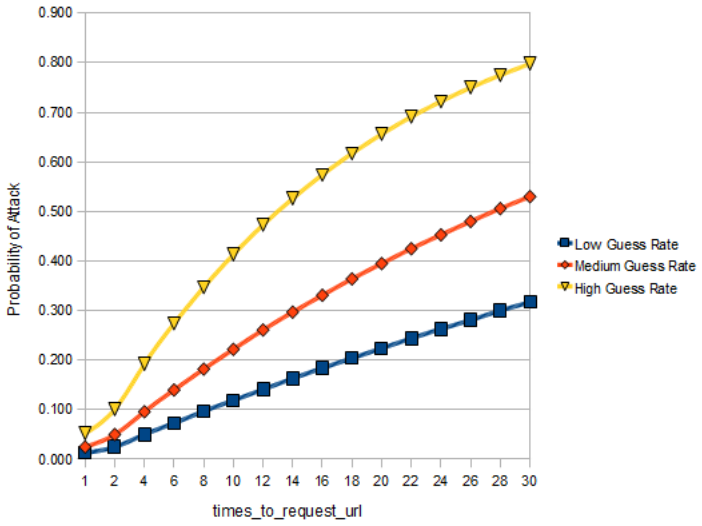

Figure 6.1: Results of varying `times_to_request_url` *with* `max_port_id` *=1*

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 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 `query_id` is given the fixed value of $2^{16} = 65536$.

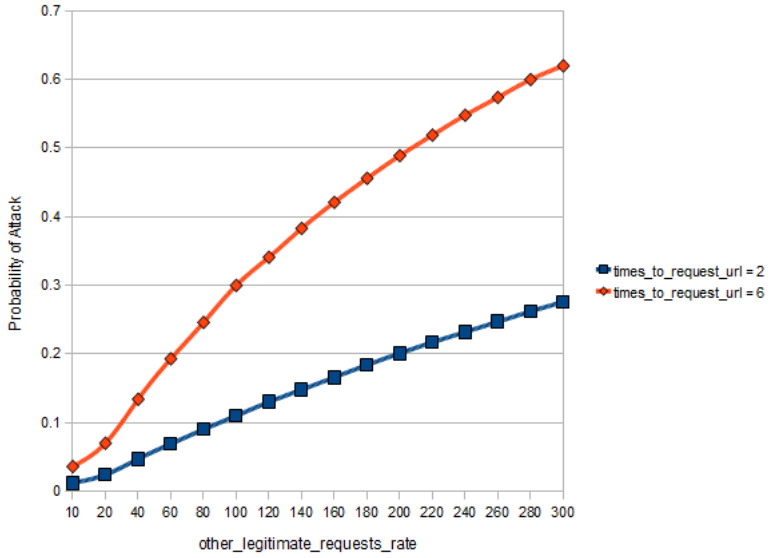

Figure 6.2: Results of varying `other_legitimate_requests_rate` *with* `max_port_id` *=1*

6.1 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. 6.1 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 url, the more opportunities there are for the IS to carry out a cache-poisoning attack. As a second-order effect, we also observe that increasing the `guess` rate increases the attack probability: the more opportunities the IS is given to guess the source-port id, the greater its probability of doing so.

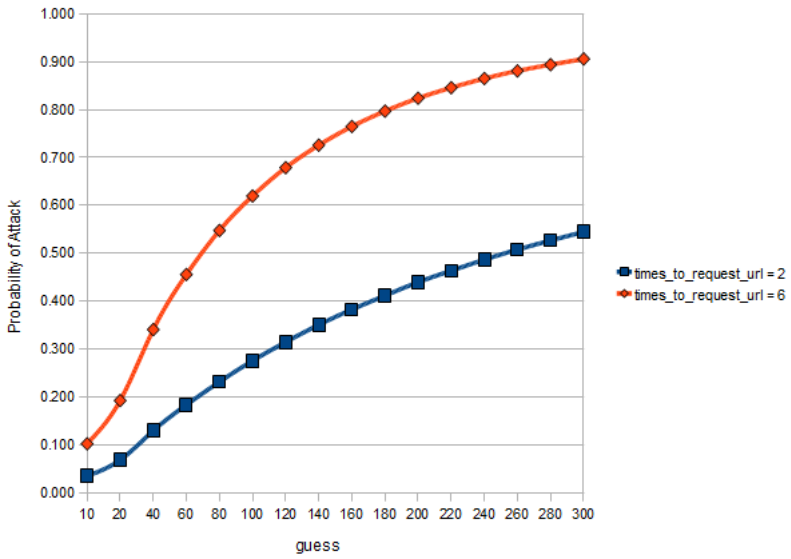

Figure 6.3: Results of varying `guess` *with* `max_port_id` *=1*

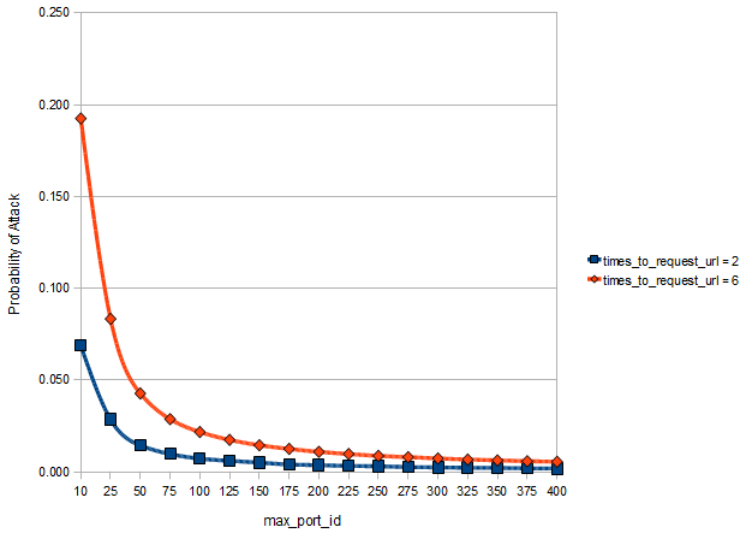

Figure 6.4: Results for different `times_to_request_url` *values while varying* `max_port_id` *values*

6.2 Result Set 2

For two different values of `times_to_request_url` (2 and 6), we vary `other_legitimate_requests_rate` from 10 to 300; the `guess` rate is set to 50 and `max_port_id` is set to 1.

As Fig. 6.2 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`.

6.3 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. 6.3 shows the positive impact of increasing the `guess` rate on the attack probability. With source-port randomization turned off (`max_port_id` =1), each guess the IS makes is correct; to poison the cache, it only needs one of these correct guesses 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.

6.4 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),

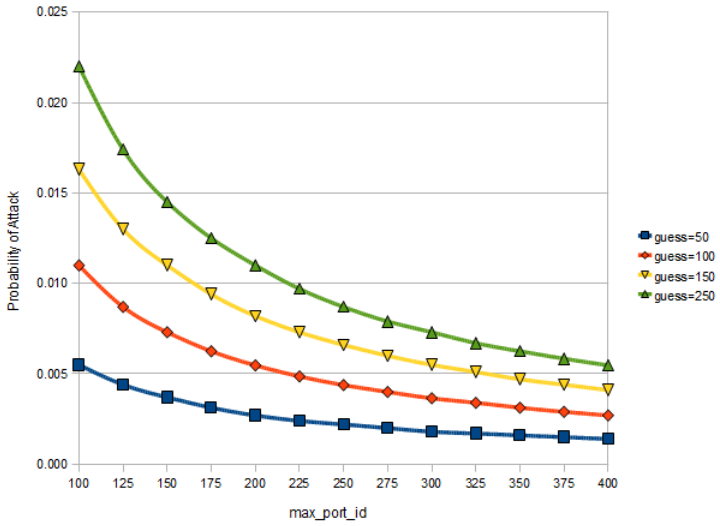

Figure 6.5: Results for different `guess` *rates while varying* `max_port_id` *values*

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. 6.4-6.6 show, the probability of a successful attack decreases exponentially as the value of `max_port_id` increases, due to the fact that the intruder now needs to guess the correct source-port id from a much larger range. Note that each of these plots follow the same basic trajectory, and exhibit a long-tail distribution. Collectively, these results validate the effectiveness of source-port randomization in countering a Kaminsky-style cache-poisoning attack.

The results of Figs. 6.4-6.6 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.

6.5 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:

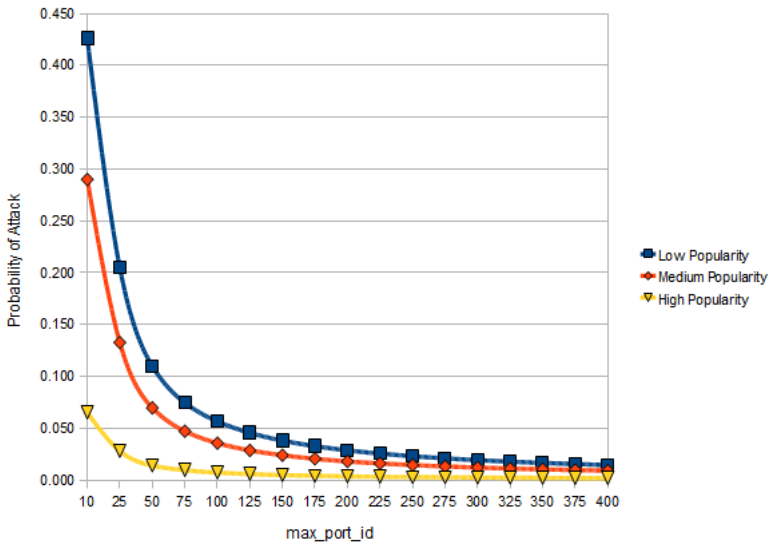

Figure 6.6: Results for different `popularity` *values while varying* `max_port_id` *values*

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` = 1 (least), `guess` = 100 (high), `other_legitimate_requests_rate` = 100 (high), and `popularity` = 0.00001 (very low), the attack probability is 0.999, which is almost 1. This result is as expected and further serves to validate the model.

Tables 6.1-6.3 contain various statistics for our CTMC model corresponding to the medium-guess-rate curve of Fig. 6.1. (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 6.1 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 6.2 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-based fixpoint algorithm [3]. The number of fixpoint iterations and the time required for them is given in Table 6.2.

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 6.1: 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 6.2: General Statistics for PRISM CTMC Model

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 6.3: Model Checking Statistics for PRISM CTMC Model

Table 6.3 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 Ubuntu 8.04. Tables 6.1-6.3 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.

Chapter 7

Related Work

In related work, a number of researchers have deployed probabilistic model checking to analyze threat levels in computer systems and security protocols [1, 2, 10, 4, 5, 14, 16, 18, 17, 19]. [17] utilizes PRISM to quantitatively analyze the probabilistic non-repudiation protocol that guarantees fairness without resorting to a trusted third party. [10] uses PRISM to estimate the probability for a malicious user breaking a non-repudiation protocol described in [17]. [18] uses symbolic model checking algorithms to automatically and efficiently construct the attack graphs for a system. Analysis of these attack graphs reveal which security vulnerabilities would be most cost-effective to guard against. The case study from [19] demonstrates how probabilistic model checking techniques can be used to formally analyze security properties of a peer-to-peer group communication system based on random message routing among members. [1] uses PRISM to analyze the anonymity provided by anonymity networks, namely Crowds, Adithia, Onion Routing, and Tarzan. [16] presents three case studies illustrating the general methodology for applying probabilistic model checking to formal verification of probabilistic security protocols. These case studies analyze Rabin's probabilistic protocol for fair commitment exchange, the probabilistic contract signing protocol of Ben-Or, Goldreich, Micali and Rivest, and a randomized protocol for signing contracts of Even, Goldreich and Lempel. [14] uses PRISM to formally model and analyze a payoff model for the reinforcement framework. PRISM can be used to model check the security of Quantum Cryptography protocols such as B92 [4] and BB84 [5].

Probably the most closely related work is that of [2], where PRISM is used to systematically quantify DoS (Denial of Service) security threats. The approach described in [2] is validated through the analysis of the Host Identity Protocol (HIP), a cryptographic key-exchange protocol with special features related to DoS protection.

To the best of our knowledge, we are the first to formally model and analyze the highly publicized Kaminsky DNS attack.

Chapter 8

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, demonstrating an exponentially decreasing, long-tail trajectory for the probability of a successful attack with an increasing range of source port ids. Conversely, our results 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. The long-term fix is DNSSEC, which prevents cache-poisoning attacks by allowing Web sites to verify their domain names and corresponding IP addresses using digital signatures and public-key encryption [7]. 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 [22]. 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/`.

Bibliography

[1] M. Adithia. Probabilistic analysis of network anonymity using prism. Master's thesis, Technische Universiteit Eindhoven, August 2006 (`http://airccse.org/journal/nsa/0410ijnsa7.pdf`).

[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] M. Elboukhari, A. Azizi, and M. Azizi. Analysis of quantum cryptography protocols by model checking. *International Journal of Universal Computer Sciences*, January 2010.

[5] M. Elboukhari, A. Azizi, and M. Azizi. Analysis of the security of bb84 by model checking. *International Journal of Network Security and Its Applications*, April 2010.

[6] 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`).

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

[8] 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.

[9] 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`).

[10] 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.

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

[12] 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).

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

[14] J. Misra and I. Saha. A reinforcement model for collaborative security and its formal analysis. In *Proc. New Security Paradigms Workshop (NSPW 2009)*, NSPW, pages 101–114. Oxford, 2009.

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

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

[17] I. Saha and D. Mukhopadhyay. Quantitative analysis of a probabilistic non-repudiation protocol through model checking. In *Proc. 5th International Conference on Information Systems Security (ICISS 2009)*, volume 5905 of *LNCS*, pages 292–300. Springer, 2009.

[18] 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.

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

[20] US CERT (United States Computer Emergenct 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).

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

[22] R. Vaughn and G. Evron. DNS amplification attacks. *DefCon Forums*, July 2006.