Stony Brook University

Formal Analysis of DNS Attacks and Their Countermeasures Using Probabilistic Model Checking

A Dissertation Presented

by

Tushar Suhas Deshpande

to

The Graduate School

in partial fulfillment of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2013

Stony Brook University

The Graduate School

Tushar Suhas Deshpande

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

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

Dr. Scott D. Stoller—Chairperson of Defense
Professor, Department of Computer Science

Dr. Radu Grosu
Associate Professor, Department of Computer Science

Dr. Panagiotis Katsaros
Assistant Professor, Department of Informatics,
Aristotle University of Thessaloniki

This dissertation is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Formal Analysis of DNS Attacks and Their Countermeasures Using Probabilistic Model Checking

by

Tushar Suhas Deshpande

Doctor of Philosophy
in
Computer Science

Stony Brook University
2013

The Domain Name System (DNS) is an internet-wide, hierarchical naming system used to translate domain names into physical IP addresses. Any disruption of the service DNS provides can have serious consequences. We present a formal analysis of two notable threats to DNS, namely *cache poisoning* and *bandwidth amplification*, and the countermeasures designed to prevent their occurrence. Our analysis of these attacks and their countermeasures is given in the form of a *cost-benefit analysis*, and is based on probabilistic model checking of Continuous-Time Markov Chains. We use CTMCs to model the race between legitimate and malicious traffic in a DNS server under attack, i.e., the victim. Countermeasure benefits and costs are quantified in terms of probabilistic reachability and reward properties, which are evaluated over all possible execution paths.

The results of our analysis support substantive conclusions about the relative effectiveness of the different countermeasures under varying operating conditions. We also validate the criticism that the DNS security extensions devised to eliminate cache poisoning render DNS more vulnerable to bandwidth amplification attacks (BAAs).

We also model the DNS BAA as a two-player, turn-based, zero-sum stochastic game between an attacker and a defender. The attacker attempts to flood the victim's bandwidth with malicious traffic by choosing an appropriate number of *zombies* to attack. In response, the defender nondeterministically chooses among five basic BAA countermeasures, so that the victim can process as much legitimate traffic as possible. We use our game-based model of DNS BAA to generate optimal attack *strategies* that vary the number of zombies and the optimal defense strategies that combine the basic BAA countermeasures to optimize the attacker's and the defender's *payoffs*. Such payoffs are defined using probabilistic reward-based properties, and are measured in terms of the attack strategy's ability to minimize the volume of legitimate traffic that is eventually processed and the defense strategy's ability to maximize the volume of legitimate traffic that is eventually processed.

To my dear mom and dad

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. Scott A. Smolka for his constant guidance and motivation. Prof. Smolka taught me how to think about a research problem and how to present its solutions in a structured manner. Prof. Smolka also helped me improve my writing skills in English. I thank Prof. Scott D. Stoller and Prof. Radu Grosu for reviewing my dissertation and providing valuable suggestions. Prof. Stoller's suggestions have been especially useful to improve our game-based model of the DNS bandwidth amplification attack. I thank Prof. Panagiotis Katsaros, without whose collaboration this work would not have been possible. Prof. Katsaros has been guiding me throughout my PhD research and he has been very instrumental in suggesting the research directions, reviewing my work, and helping me to learn the probabilistic model checking with PRISM. I thank Stylianos Basagiannis and Nikolaos Alexiou with whom I coauthored two conference papers on formal analysis of DNS cache poisoning and DNS bandwidth amplification attacks. I also thank Prof. David Parker and Aistis Simaitis, who from time to time, answered my queries on the PRISM user group. I thank Michael Cohen and John Kane, my supervisors at CA Technologies, for allowing me to work at a flexible schedule, which helped me to devote sufficient time to my research. I thank Sunil Joglekar from iGATE for motivating me to join the PhD program. I also thank Dhondopant Apte, for counseling me during the challenging times. Finally, I thank Brian Tria, the Supervising Programmer Analyst at Stony Brook University for always being there to support me. Brian also helped me to understand the working of domain name servers and to interpret their logs. I consider myself immensely fortunate for getting a friend like Brian.

Chapter 1

Introduction

DNS (Domain Name System) is a critical infrastructure component of the Internet that provides name-to-IP-address resolution; i.e., it translates domain names into physical IP addresses. DNS is implemented by a distributed hierarchical database and a query-response protocol, allowing it to scale with the growing size of the Internet. It is widely acknowledged that the initial design of DNS did not take into account the threats that came along with the immense growth of the Internet. Numerous medium- and large-scale DNS attacks have prompted the development of sophisticated countermeasures, each of which can negatively impact DNS performance and robustness (availability). Countermeasure impact on DNS deserves systematic study since the effectiveness of a countermeasure depends on the attack dynamics, and the cost associated with a countermeasure can easily outgrow the benefit it provides.

For the most prevalent threats to DNS, there is no universally deployed countermeasure. Cache poisoning attacks exploit the aggressive data caching that DNS servers use to efficiently carry out name resolution. If a DNS server's cache is eventually corrupted, users trying to access a network location are routed to a malicious IP address. The *Kaminsky attack* is a special case of cache poisoning that hijacks an entire domain [25, 35]. This causes the corrupted DNS server to reply with a malicious IP address whenever it is asked to resolve a name within the hijacked domain. At least two non-cryptographic countermeasures have been developed for cache poisoning, namely UDP (Uniform Datagram Protocol) port randomization and DNS query duplication.

Alternatively, proposed DNS security extensions [7] eliminate cache poisoning but incur significant upgrade costs and may make DNS severs more vulnerable to another important threat: *Bandwidth Amplification attack*. BAAs exploit a network of computers to flood a DNS server with excessively large DNS responses to presumed requests that have never been made. If the available bandwidth for legitimate DNS traffic is exhausted, the attack ends into a Distributed Denial of Service (DDoS) incident. Defense mechanisms that have been used include packet filtering, random drops, and aggressive retries, but various hybrid solutions may be more effective under certain circumstances.

This dissertation extends the formal analyses we previously conducted in [5] and [20] toward a versatile cost-benefit analysis framework for DNS attack countermeasures. As before, our analyses of cache poisoning and BAA are based on probabilistic model checking of Continuous Time Markov Chains (CTMCs) using the PRISM model checker [39].

Our CTMC models represent the attack dynamics and their impact on legitimate query processing by a DNS server.

Countermeasure effectiveness can be studied through different model variants, one for each countermeasure we consider, provided that costs and benefits are quantified in a consistent manner within our cost-benefit analysis framework. We define cost and benefit metrics that reflect positive and negative effects on DNS server, and compute these metrics by model checking probabilistic reachability and reward properties in the countermeasure model variants. For a parameterized countermeasure, an automated model-repair process [9] can be used to determine an optimal configuration of the parameters for a given attack probability. These are the parameter settings used in our cost-benefit analysis, which ranks countermeasures in terms of their net benefit value. Our cost-benefit analysis framework was first applied in [20] for the above-mentioned BAA countermeasures (packet filtering, random drops, and aggressive retries).

Although, the cost-benefit analysis performed using CTMCs lets us identify the most cost-effective countermeasure strategies, it requires us to first determine the optimal configuration of model parameters. Moreover, we compute the net benefit of each countermeasure separately and then, compare them to identify the most cost-effective countermeasure. In doing so, we assume that the same countermeasure is applied at all the time. However, in practice, the optimal way to apply the countermeasure may require us to use multiple countermeasures, with each countermeasure being applied for a fraction of total time. Game-theoretic modeling of DNS attacks provides a way to synthesize interesting defense strategies that combine multiple countermeasures to provide the best protection against an attack. Game-theoretic modeling also obviates the need to determine the optimal configuration of model parameters, as such configuration is automatically identified during the synthesis of optimal defense strategies. Therefore, we model the DNS BAA as a two-player, turn-based zero-sum game played between the attacker and the defender. The game is modeled using PRISM-games, an extension of the PRISM model checker that lets us model two-player, turn-based, zero-sum stochastic games and specify interesting probabilistic and reward-based properties. PRISM-games then generates the optimal strategies for a player, guaranteeing that the player can optimize a property irrespective of any strategy chosen by the adversary. In the DNS BAA game, the attacker *chooses* the number of *zombies* or the compromised machines used to launch a DNS BAA and the defender tries to prevent the attack by *choosing* the best possible countermeasures. For attacker, we specify two goals or *payoffs*, which seek to: maximize the difference between legitimate packets dropped per zombie and legitimate packets received per zombie or maximize the difference between bogus packets received and legitimate packets received. The defender chooses defense strategies that would minimize the attacker's gains. For both these properties, we record the optimal attack and defense strategies generated by PRISM-games. An optimal attack strategy varies the number of zombies used to launch the attack, while an optimal defense strategy is typically composed of two or more countermeasures.

The main contributions of this dissertation can be summarized as follows:

- We present our cost-benefit analysis framework and apply it to additional BAA countermeasures and, for the first time, to DNS cache poisoning countermeasures.

- We use our cost-benefit analysis framework to compare two main cache poisoning

countermeasures: UDP port randomization and duplicate DNS queries. Our results show that redundancy in dispatched DNS queries is less cost effective than the UDP port-randomization alternative.

- Our cost-benefit analysis framework is extended to analyze two *hybrid* BAA countermeasures, formed via a combination of packet filtering, random drops, and aggressive retries. Our results show that they are more cost effective than any one BAA countermeasure in isolation.

- We recapitulate our results of [20] for DNSSec [7], a recently proposed security extension of DNS that uses digital signatures to authenticate the source of DNS data. Together, with our new results for the hybrid BAA countermeasures, we conclude that DNSSec derives significantly less benefit from the countermeasures.

- We model the DNS BAA as a stochastic game and generate optimal attack and defense strategies.

The rest of this paper is organized as follows. Chapter 2 provides background material on the DNS cache poisoning and BAA attacks along with the countermeasures designed to prevent them. Chapter 3 introduces probabilistic model checking for CTMCs and explains how the PRISM model checker can be used to compute probabilistic reachability and reward-based properties. Chapter 4 presents our formal cost-benefit analysis framework. Chapter 5 presents our CTMC model for the DNS cache poisoning attack and the cost-benefit analysis results for the two countermeasures. Chapter 6 describes our CTMC model for the DNS BAA and the cost-benefit analysis results for the basic and hybrid countermeasures. Chapter 7 introduces stochastic game-based modeling with PRISM-games. Chapter 8 describes our stochastic game-based model of the DNS BAA. Chapter 9 discusses related work, while Chapter 10 offers our concluding remarks.

Chapter 2

DNS Attacks and Countermeasures

DNS translates user-friendly urls into the numeric IP addresses. It is implemented using hierarchically organized DNS servers. At the root of this hierarchy is the `root` DNS server, which manages top-level domain servers such as those for `com`, `edu`, and `org`. Top-level domain servers manage individual domains like `google.com`, `stonybrook.edu`, and `wikipedia.org`. Each DNS server stores the mapping of urls to IP addresses for the domains that it manages.

2.1 Domain Name System (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`. The DNS is implemented using hierarchically organized domain name servers. At the top of this hierarchy is the `root` DNS server. The root DNS server manages name servers for the top-level domains such as `com`, `edu`, and `org`. The top level domain name servers manage individual domains like `google.com`, `stonybrook.edu`, and `wikipedia.org`. The individual domain name servers such as `google.com` manage name servers for their subdomains such as `mail.google.com` and `translate.google.com`. Each domain name server maps the urls in its domain to their IP addresses.

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

Figure 2.1: 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`

Figure 2.2: Referral Response (RR) for a DNS query.

When a DNS server receives a url-resolution query from a client, typically a *web browser*, 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. 2.1. The *Question Section* contains the url to be resolved. The *Answer Section* contains the IP address for the url in the Question Section [25]. 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 [56].

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). Typically, DNS servers of the *internet service providers* (ISPs) are configured to support recursive queries. Such recursive DNS servers are also called as *DNS resolvers*. The root and other top-level domains, on the other hand, are configured to be non-recursive. A non-recursive DNS server provides a *referral response* (RRs) 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. 2.2. 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 [25].

How is a DNS query resolved? Assume that a user, who has connected his computer into the Stony Brook University's Computer Science Department's network, types the url `mail.google.com` in the web browser. The browser creates a DNS query for the domain `mail.google.com` and sends it to the ISP's DNS server, which in this case is the DNS server of the domain `cs.stonybrook.edu`. This DNS server, henceforth referred as the *local* DNS server, is a DNS resolver. So, it first searches its *cache* for the IP address of the requested domain. If the IP address for the url `mail.google.com` is found in the cache, then the cached IP address is returned to the web browser and the url resolution process concludes. However, if the IP address of the requested domain is not found in the cache, then the local DNS server tries to resolve client's query using recursion by first contacting the `root` DNS server. The `root` replies back with a RR response referring the local DNS server to the name server of the `com` domain. The local DNS server then contacts the `com` domain name server, which replies back with another RR response referring the local DNS server to the name server of the `google.com`. Finally, the local DNS server contacts the `google.com`'s domain name server, which replies with the AA response containing IP address of the `mail.google.com`. The local DNS server updates its cache with this IP address and then forwards it to the web browser.

2.2 DNS Cache Poisoning

A DNS resolver caches the IP addresses of recently resolved urls for a period of time known as the *time to live* (TTL) period. If the TTL for a url has not expired, then the cached address is used in response to all queries for that url. This in turn means that if the cache gets corrupted, i.e., if for a url an incorrect IP address is cached, then a corrupted answer is returned to the web browser. The browser then opens the incorrect web site, which could be a source of malware or a phishing site.

Let us examine the mechanics of a cache poisoning attack. Assume that an attacker wants to corrupt the cache of the DNS server for `cs.stonybrook.edu` by replacing the cache entry for `mail.google.com` with an incorrect value (IP address). In this case, `cs.stonybrook.edu`'s DNS server is the *victim* of the attack and the url `mail.google.com` is the *target* of the attack. The attacker then forces the victim DNS server to issue a DNS query for the target url [21, 5].

The DNS query is sent from a fixed UDP port, the *source port*, which is typically port 53. A unique 16-bit *query id* is associated with each DNS query, and is used to match a DNS response with the associated DNS request. While the victim waits for the correct DNS response, i.e., the DNS response containing the correct IP address of the target url, the attacker sends multiple bogus DNS responses to the victim, each of which uses a randomly generated query id and contains the IP address of a malicious web site. If the query id of one of these bogus responses matches the query id of the victim's query, then the victim accepts the bogus response, thereby corrupting its cache. The victim will subsequently respond to all requests for the target url with the corrupted cache entry.

2.3 DNS Cache Poisoning countermeasures

Full protection against DNS cache poisoning can be achieved only with cryptographic extensions to DNS such as those deployed by DNSSec [27] and DNSCurve [13]. These protocols use public-key cryptography to sign DNS records. As such, a forged DNS record is not accompanied with a valid signature and is rejected. Deploying these countermeasures, however, involves making significant changes to the existing DNS infrastructure. Therefore, the following two types of non-cryptographic countermeasures have been proposed for DNS cache poisoning.

- **UDP Port Randomization (PRAND)** Instead of using a fixed UDP port on which to send DNS queries, a randomly chosen 16-bit UDP source port is used [40]. The attacker must 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 attacker has to guess a 32-bit number [25]. The randomization level can be further increased by using 0x20-bit encoding [19], XQID [30], and WSEC-DNS [46].

- **Redundant DNS Queries (RDQ)** In this countermeasure, redundant DNS queries are used to provide protection against DNS cache poisoning. This countermeasure works as follows [57].

    1. k randomly selected ports are designated for the secure mode.

2. These k ports operate in secure mode for w seconds.
3. Any answer received on these k ports during the w-second time bound is treated as an attempted cache poisoning, and a duplicate DNS query is generated.
4. The duplicate query is resolved using the normal DNS operation. An additional table is used to keep track of the original DNS requests for which the retries are in progress. This table contains one entry per original DNS request.
5. If the answer to the duplicate query matches the answer to the first request, then the first answer is accepted.
6. If the answer to the duplicate query does not match the answer to the first request, then another duplicate query is sent.
7. Step 6 is repeated until a duplicate query returns an answer that matches the answer to the first query.
8. When the w-seconds secure-mode time window expires, the k ports resume their normal operation, i.e., they are no longer in a secure mode and do not send duplicate queries.
9. Steps 1- 8 are repeated with a new set of k randomly chosen ports.

Since a response is cached only if both the original query and the duplicate query receive identical responses, an attacker has to guess the correct <query id, port id> twice, thereby reducing the likelihood of an attack.

Stronger security is obtained with higher values of k and w. It is, therefore, reasonable to set k to the number of all available UDP ports and w to *infinity*; i.e., the secure mode is enabled for *all* available ports *all* the time.

Although both PRAND and RDQ mitigate DNS cache poisoning, they cannot fully eliminate it, as there exists no built-in mechanism within DNS for establishing trust between a DNS server issuing a query and a DNS server responding to that query. The DNS Security Extensions (DNSSec) use digital signatures and public-key encryption to authenticate the authoritative DNS servers [27]. A DNSSec response is accepted only if it is verified to have come from an authenticated server, thereby eliminating any possibility of cache poisoning. DNSSec, however, requires significant infrastructure upgrades. Also, because of the increased request and response size, DNSSec may be more vulnerable to DDoS attacks, the most prominent of which is BAA. Therefore, we need to study DNSSec's behavior in presence of BAAs to determine if DNSSec is more vulnerable than DNS to a BAA, and if we have effective countermeasures for BAAs.

2.4 DNS Bandwidth Amplification Attack (BAA)

The DNS Bandwidth Amplification Attack (BAA) is a distributed denial-of-service attack in which a network of computers floods a DNS resolver with large responses to requests that have never been made. A typical DNS response size is 512 bytes. However, during BAA, the victim DNS resolver can receive DNS responses that are as large as 4000 bytes. These unwanted responses consume both the bandwidth and the computational power of

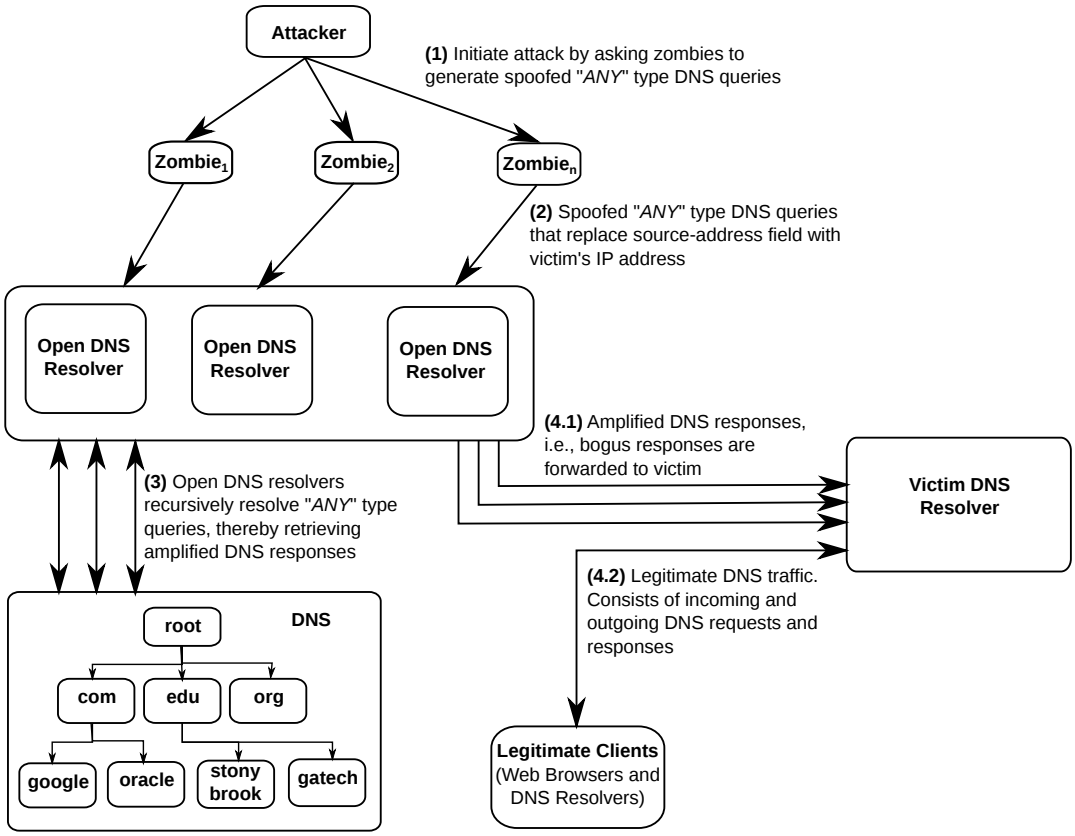


Figure 2.3: Schematic diagram of DNS BAA

the victim DNS server. The BAA is a major cause of the DNS disruption; a number of incidents involving BAA have been reported since 2002.

How to generate amplified DNS responses? Typical DNS queries, such as the ones described in section 2.1 simply seek to know the IP address for a desired url. Such queries are known as "*A*" type queries [4]. The responses to *A* type DNS queries are average sized, because they contain only the required IP addresses.[1] However, it is also possible for a client to issue another kind of DNS requests, the "*ANY*" type requests. They ask a DNS server to return SOA (Start of Authority) record for the domain, the IP addresses of various name servers for the domain, the IP addresses of the mail servers for the domain, and IP addresses for desired urls. If the security extension of DNS, the DNSSec, is enabled, such responses also contain cryptographic signatures (RRSIG records) associated with name servers. So, the size of a response to an *ANY* type query can be much larger than size of a response to an *A* type query. Using *ANY* type queries, it is possible to generate DNS responses that are 50 times larger than the request size. So, the *amplification factor* (AF), i.e., the ratio of the response size to the request size is 50 [48]. For DNS, the AF can be as high as 73 [36, 58], while the AF for DNSSec can be as high as 271.2 [18]. The fact that a small sized DNS request can generate a substantially larger response makes the BAA possible.

[1]The AA and RR responses in Figs. 2.1 and 2.2 are examples of the responses generated for *A* type queries.

Let us now understand how an attacker can launch a DNS BAA. Let us assume that the attacker decides to launch a BAA on the DNS resolver for the domain `cs.stonybrook.edu`, henceforth referred as the victim server. Fig. 2.3 depicts a BAA.

1. The attacker prepares for the BAA by
   - Acquiring control of a large number of compromised hosts (zombies) to be used as attack sources.
   - Acquiring a list of open DNS resolvers. A DNS resolver is *open* if it is able to provide recursive name resolution service for clients outside of its administrative domain [22]. Typically, DNS resolvers are configured to answer requests from only those clients that are within their own domain [47]. For example, the DNS resolver of `cs.stonybrook.edu` can answer DNS queries originating from machines that are connected to Stony Brook University's Computer Science Department's network, however, it would reject queries generated by clients that are in some other domain, such as `google.com`. Open DNS resolvers, on the other hand, accept queries from any DNS client. Because of this property, open DNS resolvers are used in BAAs.
2. The attacker commands the zombies to send the previously found open DNS resolvers a number of requests that would generate amplified responses (Step 1 from Fig. 2.3). E.g., the zombies may send a number of *ANY* type requests to the open DNS resolvers (Step 2 from Fig. 2.3). Moreover, these queries are *spoofed*, i.e., the source-address fields of these queries have been replaced with the victim's IP address. The open DNS resolvers resolve the spoofed queries (Step 3 from Fig. 2.3) and direct the large number of amplified response that they receive to the victim server (Step 4.1 from Fig. 2.3), thereby exhausting victim's available bandwidth.

2.5 DNS BAA countermeasures

Three basic countermeasures are suggested to prevent the DNS BAA.

- **Filtering (FTR):** Filtering tries to identify and block the attack traffic. It offers relatively high accuracy with a false-positive rate as low as 10% [55, 45, 33, 31, 64]. The computational demands of FTR depend on the filtering mechanism and the attack strength.
- **Random Drops (RND):** RND regulates incoming traffic by randomly dropping the DNS packets [41, 32]. During a BAA, the traffic arriving at the victim mainly comprises the bogus packets, and a randomly dropped packet is, therefore, likely to be bogus. RND has negligible computational demands.
- **Aggressive Retries (AGR):** AGR encourages the legitimate clients to generate traffic at a higher rate. In AGR, increasing the number of retries by one doubles the amount of legitimate traffic generated during each retry [59, 37]. The downside is the increased server workload and the increased bandwidth consumption.

It is possible to combine the basic BAA countermeasures to get a better protection.

- **Random Drops with Aggressive Retries (RDR):** RDR is obtained by combining RND with AGR.
- **Aggressive Retries with Filtering (AGF):** By combining AGR with FTR we get the AGF.

Both RDR and AGF try to filter out the attack traffic while explicitly increasing the proportion of the legitimate traffic.

Chapter 3

Probabilistic Model Checking using CTMCs

Our cost-benefit analysis of countermeasures aimed at preventing cache poisoning and bandwidth amplification attacks on DNS servers is based on CTMC models of these attacks. CTMCs provide a natural modeling formalism in which the arrival processes for benign and malicious server requests and, concomitantly, the race between these two types of requests can be represented. The resulting probabilistic behavior can be analyzed using *probabilistic model checking*, a highly successful automated analysis technique.

CTMC-based modeling inherently involves the *fundamental Markov property* assumption: the conditional probability distribution of future model states depends only on the present state. This implies that the waiting time for transitions is governed by a negative exponential distribution, and, consequently, transition times occur according to a Poisson process. Poisson distributions are commonly used to model arrival processes for client requests in client-server systems, such as the processes generating DNS traffic.

For BAA, we note that the validity of the Markov property for representing bandwidth sharing has been shown in [24]. Bandwidth sharing, under the common assumption of Poisson session arrivals, is insensitive to the flow size and the packet arrival process. We can, therefore, reasonably assume that the conditional probability distribution of future states depends only on the present state.

Definition A *labeled CTMC* is a tuple $C = (S, \overline{s}, R, L)$ where:

- S is a finite set of *states*;
- $\overline{s} \in S$ is the *initial state*;
- $R : S \times S \to \mathbb{R}_{\geq 0}$ is the *transition rate matrix*;
- $L : S \to 2^{AP}$ is a *labeling function* which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state.

The transition rate matrix R associates a rate with each pair of states in the CTMC. A transition can occur from state s to state s' if $R(s, s') > 0$. The probability of this transition taking place within t time units is $1 - e^{-R(s,s') \cdot t}$. The time spent in state s before a transition

taking place is exponentially distributed with rate $E(s)$, where $E(s) = \sum_{s' \in S} R(s, s')$. An execution of a labeled CTMC $C = (S, \overline{s}, R, L)$ is represented by a path. Formally, a path ω is a non-empty sequence of states $s_0 s_1 s_2 \ldots$ where $s_i \in S$ and $P(s_i, s_{i+1}) > 0$ for all $i \geq 0$.

Probabilistic model checking is an automated formal verification technique for modeling and analyzing systems or processes with probabilistic behavior, e.g., the CTMC of a queuing system. Model checking tools like PRISM [39] involve a combination of graph-theoretic algorithms for reachability analysis and iterative numerical solvers. Thus, it is possible to evaluate properties of the form $\mathcal{P}_{=?}(\psi)$ that compute the probability of some path that satisfies ψ. Path formula ψ is interpreted over the paths of the probabilistic model, which could be a Discrete Time Markov Chain (DTMC), a Continuous Time Markov Chain (CTMC), or a Markov Decision Process (MDP). We typically define properties of the form `F prop`, where `F` is the "eventually" linear temporal operator and `prop` is a state assertion that evaluates to true or false for a single model state.

PRISM also allows us to assign rewards to *states* and *transitions*, such that they accumulate over time. If at time t the model has reached the *n* th state of some path, the *cumulative* reward is the sum of rewards accumulated in the preceding states (or transitions). Reward can be also interpreted as cost. PRISM lets us write reward-based properties, which compute *expected cumulative rewards*. These properties can be used to compute measures such as expected number of legitimate packets lost. A reward-based property in PRISM is of the form $\mathcal{R}\{\text{“}\texttt{rewardId}\text{”}\} =?[\psi]$, where the $\mathcal{R}$ operator signifies a reward-based property, `rewardId` is the identifier representing the reward structure to be used, and ψ is a path formula. This property, upon its evaluation, would return the expected reward accumulated for the reward structure `rewardId` until path property ψ is satisfied.

A model in PRISM is constructed as the parallel composition of its modules, where each module consists of *variables* and *commands*. The variables represent possible states for the module. The commands describe the module's behavior, i.e., the way in which the module's state evolves over time. Each command comprises a guard and one or more update actions:

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

Guard `g` is a predicate over model variables, whereas each update $\texttt{u}_i$ describes, by assigning new values to the variables, a transition that the module can make. For CTMCs, λ_i is the transition's rate, the parameter of a negative exponential distribution that governs the waiting time of the transition. If the guard is true, the updates are executed according to their rates. Commands can be labeled and this allows modules to interact with each other by synchronizing on identically labeled commands. In this case, the rate of the resulting transition is the product of the rates of the individual transitions.

Building a CTMC model with PRISM involves identifying model variables, model actions, and rates for the actions. Model parameters are the variables that can take a range of values. By varying the values of model parameters, their impact on model behavior can be ascertained. In contrast, model constants are the variables that are assigned a fixed value.

PRISM also provides the $\mathcal{R}$ operator to analyze reward-based properties. Four different types of reward-based properties are supported, but, for our analysis, we use only

cumulative reward properties of the form ($\mathcal{C} \leq t$). These are appropriate for evaluating the effects of BAA countermeasures in the small time period during which the attack has to be mitigated.

Chapter 4

Cost-Benefit Analysis via Probabilistic Model Checking

Cost-benefit analysis is a technique for evaluating an activity by comparing its benefits with its costs [51]. A cost-benefit analysis serves two purposes. First, it is used to determine the viability of an activity and secondly, it is used to compare the performance of available alternatives. For a given activity, assume that we have identified benefits $B_1, B_2, ..., B_n$ and costs $C_1, C_2, ..., C_k$. Then, the net benefit is computed as

$$net\ benefit = \sum_{i=1}^{n} B_i - \sum_{j=1}^{k} C_j \tag{4.1}$$

Benefits are measures of improvements caused by the activity in question, whereas costs represent undesired side-effects. For example, consider packet filtering, which acts as a deterrent against DNS BAA attacks and reduces the attack probability by filtering bogus traffic. Packet filtering, however, also has false positives; i.e., it filters out some of the legitimate traffic. So, we define a benefit metric for packet filtering that reflects the increase in the percentage of the legitimate traffic in the total traffic, and a cost metric that reflects the false-positive rate.

In a cost-benefit analysis, it is possible to assign *weights* in order to take into account the relative importance of benefits and costs [34]. While computing the net benefit, each benefit and cost metric is multiplied by its weight [14]. Although weighted cost-benefit analysis can give more accurate results, the determination of suitable weights is specific to the needs and policies of the organization [54]. For this reason, in our cost-benefit analysis, we assume all benefits and costs to be equally important, and do not include weights in the cost-benefit analysis. Cost and benefit metrics are computed in our framework by model checking appropriate probabilistic reachability and reward properties.

Cost and benefit metrics are selected so that they satisfy the following criteria [60].

1. **Common unit of measurement:** In performing a cost-benefit analysis, we compute the net benefit using Equation (4.1). This is possible only if all benefit and cost metrics are expressed in a common unit. A practical approach is to represent all metrics as unit-less quantities such as percentages.

2. **Avoidance of double counting:** Often, the impact of an activity can be measured in more than one way; e.g., in a typical attack scenario, the effectiveness of a countermeasure can be measured as the reduction in the attack probability, the increase in the time needed for the attack to succeed, or the increase in the attacker's effort. Including all these metrics as benefits in a cost-benefit analysis would count the same effect multiple times. In such cases, care must be taken so that all countermeasure effects are represented in the cost-benefit analysis by a single metric. Statistical techniques like the Pearson product-moment correlation coefficient [61] help us to detect highly correlated measurements of candidate metrics, in order to avoid double-counting.

Our cost-benefit analysis aims to evaluate and compare the effectiveness of different countermeasures designed to protect a DNS server. For this reason, selected metrics should cover all countermeasure effects on DNS robustness and performance, but should not refer to the usage of resources for implementing a countermeasure (e.g., CPU time, memory usage). Thus, our cost-benefit analysis cannot be directly used for economic evaluation of the countermeasures, where multiple concerns have to be taken into account.

In carrying out our cost-benefit analysis, we identified two cases. 1) The countermeasures under consideration are controlled by the same parameters. This is the case for DNS cache poisoning (see Chapter 5): both countermeasures depend on the same parameter, `port_id_bits`. 2) The countermeasures have different parameters; this is the case for the BAA countermeasures (see Chapter 6).

In the first case, the common parameters are systematically varied to compute costs, benefits, and associated net benefit values. If for certain parameter values a countermeasure's $net\ benefit > 0$, then the countermeasure is profitable for those cases.

In the second case, we identify the parameter settings that produce a desired value for a particular quantity, such as the desired attack probability. This approach can be seen as a version of the *model repair* problem for probabilistic systems considered in [9]. With countermeasure parameter values determined in this manner, we then evaluate other probabilistic and reward-based properties, which are used to compute benefits, costs, and net benefits. This allows us to compare the five BAA countermeasures we consider in Chapter 6, each of which is controlled by different parameters, thereby identifying the countermeasure that offers the highest net benefit while achieving the desired attack probability.

Our cost-benefit analysis concludes with a comparison of the net benefits observed for properly selected model parameter values. A countermeasure offering the maximum net benefit is the best performing countermeasure among all alternatives.

Chapter 5

Cost-Benefit Analysis for DNS Cache Poisoning Countermeasures

We present the PRISM CTMC model for the DNS cache poisoning attack and its two countermeasures. We use this model to compute the net benefit for the two countermeasures. (*Note:* Appendix A contains the fully documented model code)

5.1 CTMC model of DNS Cache Poisoning countermeasures

Our model defines five modules.

- **Client Server (CS):** CS is the victim of the attack. It is recursive, maintains a cache, and is authoritative for the domain `cs.stonybrook.edu`. If a url is successfully resolved, then its IP address is stored in the CS's cache until the TTL expires.
- **Client Machine (CM):** CM is a normal desktop or a laptop computer that is served by the CS. CM begins query resolution process by asking the CS to resolve the url `mail.google.com`.
- **Non-Authoritative Servers (NAS):** NAS represents all intermediate non-authoritative DNS servers that are involved in the url resolution process. NAS always return referral responses asking the requester to query some other DNS server that is more likely to know the IP address of the requested url. The NAS module collectively represents the `root` DNS server and the `com` DNS server.
- **Domain Server (DS):** DS is the authoritative name server for the target domain (`google.com`). It returns an authoritative response containing the IP address for any url within the target domain.
- **Attacker Server (AS):** AS is the authoritative DNS server for the domain `badguy.com`. This domain is controlled by the attacker.

The model parameters are the following:

- `port_id_bits`: Defines the range of the number of port id bits as 1..`port_id_bits`. The number of available ports is $2^{\texttt{port\_id\_bits}}$.

- `guess`: The overall rate at which the AS sends bogus responses to the CS. These responses may be correct or incorrect guesses, depending on if the attacker was able to correctly guess the source-port id.

- `popularity`: The rate at which the TTL associated with the CS's cache entry for `mail.google.com` has a positive value. The more popular a 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.

- `other_legitimate_requests_rate`: The rate at which requests from DNS servers other than CS arrive at the DS. The parameter `other_legitimate_requests_rate` is therefore used to represent the load on the DS. If the load on the DS is high, then it takes more time to process the DNS requests and return the responses.

- `NAS_count`: Represents the number of non-authoritative servers (NASs) involved in the url resolution. The minimum value of `NAS_count` is 1, since at least one NAS, the `root` name server, is involved in resolution of any url. `NAS_count` determines the number of recursive queries generated by CS. CS generates one recursive query for each NAS. Additionally, one recursive query is generated when CS communicates with DS. Therefore, the number of recursive queries generated by CS is `NAS_count` + 1. The higher the `NAS_count`, the longer the CS has to wait to obtain the correct IP address for a requested url. This gives an attacker more time to attack, thereby increasing the attack probability.

Each module defines certain actions that 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 guards 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 AS

The attacker has two opportunities for corrupting the CS's cache: when the CS is waiting for a referral response from the NAS, and when the CS is waiting for the authoritative response from the DS. This is reflected in the following actions.

a. **Send correct guess to CS while in race with DS:** The AS sends a bogus response to the CS that correctly matches the CS's source-port id, thereby poisoning its cache. This action synchronizes with action **Receive correct guess from AS while in race with DS** of CS, and has an associated rate given by parameter `guess`.

b. **Send correct guess to CS while in race with NAS:** The AS sends a bogus response to the CS that correctly matches the CS's source-port id, thus poisoning its cache. This action synchronizes with action **Receive correct guess from AS while in race with NAS** of CS, and has an associated rate given by parameter `guess`.

Actions Defined for CS

a. **Send url-resolution request to NAS:** With rate $1 - \texttt{popularity}/10$, the TTL of the target url's cache entry is set to 0. In this case, the requested url does not exist in the cache, and a query is sent to the NAS. With rate popularity/10, the TTL is set to 1. In this case, the IP address for the requested url is cached, and query is marked as answered. This action synchronizes with action **Process request sent by CS** of NAS.

b. **Receive response from DS:** In this case, the DS has won the race with the AS and cache poisoning has been avoided. This action is synchronized with action **Process request sent by CS** of DS. 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 AS while in race with DS:** Let n = max_query_id · max_query_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 while in race with DS** of AS. The combined arrival rate for correct guesses is obtained by multiplying the rates of these two synchronizing actions: $(1/n) \cdot \texttt{guess}$.

d. **Receive correct guess from AS while in race with NAS:** This action executes with rate $1/n$, where n = max_query_id · max_query_id and synchronizes with action **Send correct guess to CS while in race with NAS** of AS. The rate for this action is similar to the rate for the **Receive correct guess from AS while in race with DS**.

Action Defined for NAS

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 the CS. This action synchronizes with action **Send url-resolution request to NAS** of CS and executes at the rate of 1/(NAS_count).

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 the CS. The rate for this action is given by 1/other_legitimate_requests_rate.

We now describe how the basic CTMC model of DNS cache poisoning can be extended to analyze the two countermeasures PRAND and RDQ.

- **Modeling PRAND:** PRAND is implemented by varying the parameter port_id_bits. We vary port_id_bits from 1 to 16. Setting port_id_bits = 0 has the effect of turning off port randomization.

- **Modeling RDQ:** We can see that each duplicate query is an instance (execution) of PRAND. Therefore, RDQ can be analyzed from the results of PRAND. We present the formulas for computing the attack probability after the n^{th} retry. For PRAND, with a given attack setting (`guess`, `port_id_bits`), let p be the attack probability and $q = (1 - p)$ be the probability of obtaining the correct target IP address. Let n be the number of times a duplicate query is sent. We consider the following cases.

  - $\mathbf{n = 1}$ The probability of an attack during the *first* retry is the probability of an *attack* followed by an *attack*, which is $p \cdot p = p^2$.
  - $\mathbf{n = 2}$ The probability of attack during the *second* retry is the probability of an *attack* followed by a *failed attack* followed by an *attack*, which is $p \cdot q \cdot p = p^2 \cdot q$.
  - $\mathbf{n = 3}$ The probability of attack during the *third* retry is the probability of an *attack* followed by a *failed attack* followed by a *failed attack* followed by an *attack*, which is $p \cdot q \cdot q \cdot p = p^2 \cdot q^2$.

  Generalizing, we obtain

$$\text{probability of attack } \mathbf{during} \text{ the } n^{th} \text{ retry} = p^2 \cdot q^{(n-1)} \tag{5.1}$$

  Similarly, we can derive

$$\text{probability of no attack } \mathbf{during} \text{ the } n^{th} \text{ retry} = q^2 \cdot p^{(n-1)} \tag{5.2}$$

  Now, let us determine the probability of attack and the probability of no attack, i.e., a failed attack, *after* the n^{th} retry.

$$\begin{aligned} &\text{probability of attack } \mathbf{after} \text{ the } n^{th} \text{ retry} \\ &= \text{probability of attack } \mathbf{during}(\text{first retry } OR \text{ second retry} \\ &\ OR \text{ third retry } OR \ldots OR \ n^{th} \text{retry}) \\ &= p^2 + p^2 \cdot q + p^2 \cdot q^2 + \cdots + p^2 \cdot q^{(n-1)} \\ &= p \cdot (1 - q^n) \end{aligned} \tag{5.3}$$

  We can similarly determine

$$\text{probability of no attack } \mathbf{after} \text{ the } n^{th} \text{ retry} = q \cdot (1 - p^n) \tag{5.4}$$

  It is possible, however, that in spite of performing n retries we may not get an answer that matches the answer to the first request. In this case, we are required to perform the $(n + 1)^{st}$ retry.

$$\begin{aligned} &\text{probability of another retry required } \mathbf{after} \text{ the } n^{th} \text{ retry} \\ &= 1 - (p \cdot (1 - q^n) + q \cdot (1 - p^n)) \\ &= p \cdot q \cdot (p^{n-1} + q^{n-1}) \end{aligned} \tag{5.5}$$

  Since $p < 1$ and $q < 1$, from Equations (5.3)-(5.5) we observe that with increasing n, the probability of attack after the n^{th} retry tends to p, the probability of no attack

after the n^{th} retry tends to q, and the probability of another retry required after the n^{th} retry tends to *zero*. This suggests that if we have identical attack settings for RDQ and PRAND, then in RDQ, an attacker has to perform a greater number of guesses to achieve the attack probability that is equal to the attack probability p observed for PRAND.

Two variants of RDQ: Depending on the restrictions imposed on n, we can identify two variants of the RDQ.

1. **Unrestricted *n* ($\mathbf{RDQ_1}$):** As seen in Chapter 2.2, when RDQ is deployed, a new url-resolution request received from the client machine is resolved normally yielding an IP address. This answer, however, is not trusted and a duplicate query is generated. In RDQ_1, we do not impose any upper limit on n. Therefore, duplicate queries are repeatedly generated until the most recent duplicate query returns a response that matches the untrusted response obtained by resolving the client's original query. For a given attack setting, we then determine the expected value of n. The expected value of n can be computed as follows. From Equation (5.5), we observe that the probability that another retry is required after the n^{th} retry is $p \cdot q \cdot (p^{n-1} + q^{n-1})$. We find the value of n for which $p \cdot q \cdot (p^{n-1} + q^{n-1}) = 0$. Let us denote this value of n as the n_{max}. This ensures that after n_{max} retries, we would obtain an answer that matches the answer received for the original query. We can then find the expected value of n, the $n_{expected}$, by evaluating

$$n_{expected} = \sum_{i=1}^{n_{max}} i \cdot (p \cdot q \cdot (p^{i-1} + q^{i-1})) \tag{5.6}$$

   For given values of p and q, both n_{max} and $n_{expected}$ can be evaluated by running a simple script, a Groovy version of which is available in Appendix A.

2. **Restricted *n* ($\mathbf{RDQ_2}$):** We impose an upper limit, n_{max}, on the number of times duplicate queries are generated. We restrict n_{max} to the range $1 \ldots 4$ and compute $n_{expected}$. For identical attack settings, let $n'_{expected}$ be the value of $n_{expected}$ for RDQ_1. Then, $n_{expected}$ for RDQ_2 is computed by evaluating the following conditional expression.

$$n_{expected} = (n'_{expected} \ < \ n_{max}) \ ? \ n'_{expected} \ : \ n_{max} \tag{5.7}$$

5.2 Benefit and cost metrics

- **For PRAND**

  The attack probability is the probability of the victim receiving a bogus response from the AS before the victim receives the correct response from the DS. In PRAND, as we increase the value of `port_id_bits`, we expect the attack probability to decrease. The attack probability, p, is computed by evaluating the Continuous Stochastic Logic (CSL) formula `P=? [F "attacked"]`, where the state assertion `"attacked"` becomes true when the AS correctly guesses the victim's

source-port id. Let p be the attack probability observed for a given attack setting, and p_0 be the attack probability observed with no port randomization, i.e., with `port_id_bits` = 0. Then, we define the percentage decrease in the attack probability ($p_0 - p$), over p_0 as the benefit metric B_1:

$$B_1 = \frac{p_0 - p}{p_0} \cdot 100. \tag{5.8}$$

For PRAND, the net benefit is simply

$$net\ benefit = B_1. \tag{5.9}$$

- **For RDQ**
  Benefit B_1 from equation (5.8) is also applicable to RDQ, and the value of p is computed using equation (5.3). The expected number of retries, $n_{expected}$, give rise to a new cost C_1 by increasing CS's bandwidth usage. We define C_1 as the percentage increase in CS's bandwidth usage over its bandwidth usage when RDQ is not deployed. The bandwidth usage depends directly on the number of retries, so

  $$C_1 = n_{expected} \cdot 100. \tag{5.10}$$

  The number of retries performed also increase the response time for CS. It can be easily observed, however, that both the bandwidth usage and the response time directly depend on $n_{expected}$. Therefore, in the cost-benefit Analysis, we include a cost metric based on increased bandwidth usage alone. Another cost metric, C_2, is associated with RDQ. C_2 is the percentage representation of the probability of the query not getting resolved despite performing the $n_{expected}$ retries. This is possible if $n_{expected} < n_{max}$, which is true for low values of `port_id_bits`. Using Equation (5.5), we can compute

  $$C_2 = p \cdot q \cdot (p^{n_{expected}-1} + q^{n_{expected}-1}) \cdot 100. \tag{5.11}$$

  For RDQ, the net benefit is thus computed as

  $$net\ benefit = B_1 - C_1 - C_2. \tag{5.12}$$

5.3 Experimental results

Because PRAND and RDQ have the same control parameter, viz., `port_id_bits`, we do not follow a model-repair-based approach to find optimal parameter settings for a given attack probability. We instead vary `port_id_bits` and record the observed costs and benefits. This lets us compare how the two countermeasures perform for different values of `port_id_bits`.

For a domain with medium popularity (`popularity` = 5), handling a moderate `other_legitimate_requests_rate` of 100, we apply both PRAND and RDQ. We vary `port_id_bits` from 1 to 16 and record the net benefits computed using Equations (5.9) and (5.12). Then, for three different values of the `guess` rate – 10000, 100000,

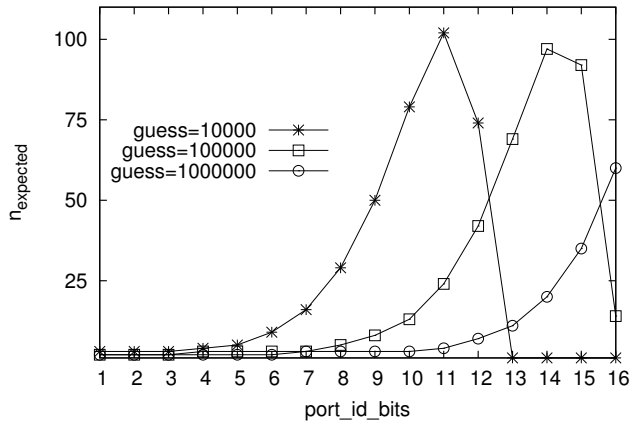


Figure 5.1: $n_{expected}$ computed using Equation (5.6) *for* RDQ_1

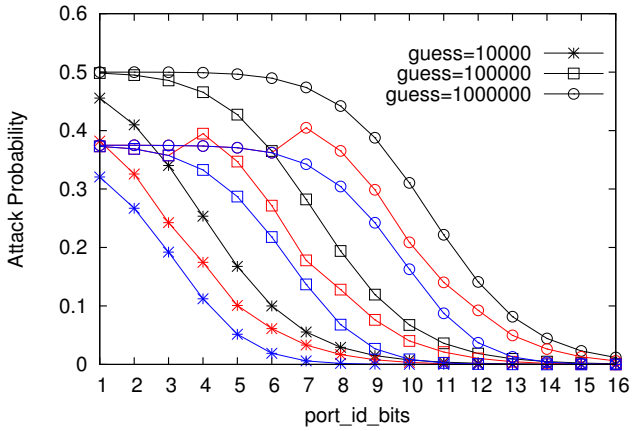


Figure 5.2: Attack probability for PRAND and RDQ at different `guess` *rates for varying number of* `port_id_bits`*. [Color codes: black - PRAND, red -* RDQ_1*, and blue -* RDQ_2*]*

and 1000000 – we plot the net benefit against `port_id_bits`. We assume that on average CS generates five recursive queries in resolving a url. Therefore, `NAS_count` is set to 4.

Since RDQ encompasses PRAND's behavior, the attack probability p in RDQ tends to *zero* as `port_id_bits` increases. In such cases, according to Equation (5.6), RDQ_1 is required to perform *zero* retries. In both variants of RDQ, however, at least one retry is always performed. So, for RDQ_1, $n_{expected}$ tends to 1 as `port_id_bits` increases. In general, n varies between 1 and 25, sometimes increasing to 100 (see Fig. 5.1). So, for RDQ_2, we set $n_{max} = 2$. Henceforth, we refer to RDQ_2 with $n_{max} = 2$ as simply RDQ_2.

Fig. 5.2 plots the attack probability observed for PRAND and RDQ at different `guess` rates versus `port_id_bits`. The black plot shows the attack probability observed for PRAND, the red plot shows the attack probability observed for RDQ_1, and the blue plot shows the attack probability observed for RDQ_2. We observe that the attack probability increases with increasing `guess` rate. This shows that the more guesses an attacker generates, the higher is the probability of the attack being successful. We also see that the attack

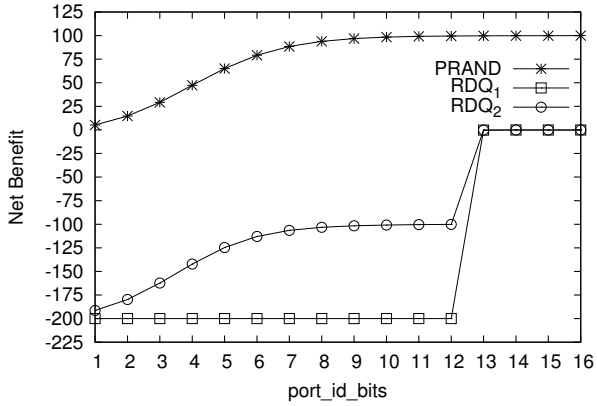


Figure 5.3: Net benefit for PRAND and RDQ for varying number of `port_id_bits` *with* `guess` *= 10000.*

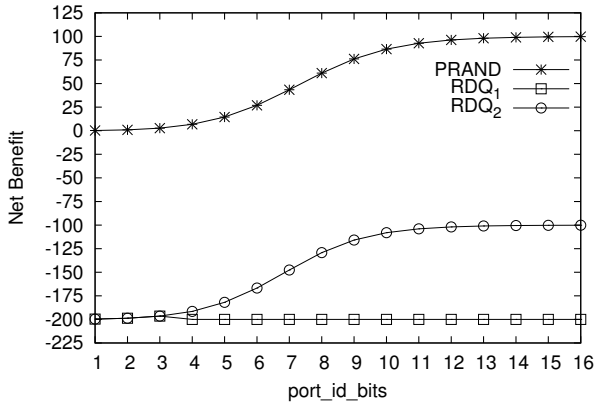


Figure 5.4: Net benefit for PRAND and RDQ for varying number of `port_id_bits` *with* `guess` *= 100000.*

probability observed for both variants of RDQ is always less than the attack probability observed for PRAND. This supports the observation of [57] that RDQ offers superior protection than PRAND against DNS cache poisoning. We see that for lower values of `port_id_bits`, RDQ_2 offers a lower attack probability than the attack probability observed for RDQ_1. This happens because, as seen from Fig. 5.1, for lower `port_id_bits`, RDQ_1's $n_{expected} \geq 2$. We also recall that for RDQ_2, $n_{max} = 2$. So, according to Equation (5.3), RDQ_2 offers a lower attack probability than the attack probability observed for RDQ_1.

Figs. 5.3-5.5 plot the net benefits for PRAND and RDQ versus `port_id_bits` at varying guess rates. From our experiments, we observe that the net benefit for RDQ_2 is always greater than -200, since for RDQ_2, $n_{max} = 2$. As seen from Fig. 5.1, however, for some values of `port_id_bits`, RDQ_1 can have a large $n_{expected}$, thereby causing its net benefit to fall below -200. In order to improve the readability of Figs. 5.3-5.5, we limit the negative net benefit at -200.[1] This ensures that we plot observed net benefit values for RDQ_2, while eliminating highly negative net benefit values for RDQ_1. Figs. 5.3-5.5 show

[1]This technique is similar to the techniques used to *discretize* data during data mining [29].

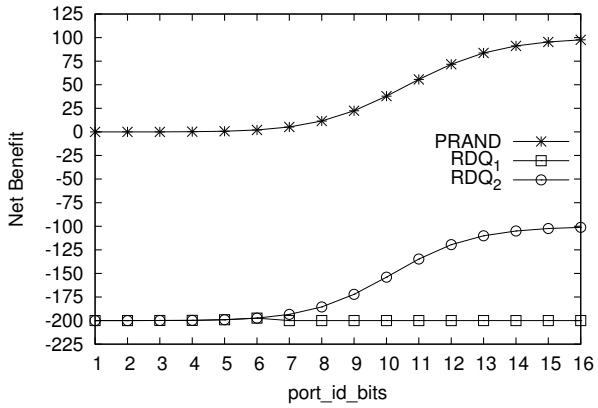


Figure 5.5: Net benefit for PRAND and RDQ for varying number of `port_id_bits` *with* `guess` *= 1000000.*

`port_id_bits`	PRAND	RDQ_1			RDQ_2		
	B_1	B_1	C_1	C_2	B_1	C_1	C_2
1	0.23	25.33	200	24.99	25.33	200	24.5
4	6.78	21.01	200	12.5	33.4	300	24.88
7	43.49	64.38	200	12.05	72.6	300	20.26
10	86.5	91.95	200	2.72	98.24	1300	6.29
13	98.1	99.08	200	0.49	99.96	6900	0.94
16	99.76	99.99	200	0.12	99.99	1400	0.12

Table 5.1: Costs and benefits for PRAND and RDQ, with `guess` *= 100000*

that both variants of RDQ offer lower net benefit than the net benefit offered by PRAND.

Table 5.1 shows the cost and benefit values for PRAND and RDQ when `guess` = 100000. We see that both variants of RDQ offer higher B_1 than PRAND. For RDQ, however, the increase in B_1 is insufficient to overcome the cost ($C_1 + C_2$). So, the net benefit of RDQ is less than the net benefit of PRAND.

5.4 Observations

From the cost-benefit analysis results presented in section 5.3, we observe that PRAND always performs better than RDQ. We would like to point out, however, that the cost-benefit Analysis results could be different if a weighted cost-benefit Analysis were used. From Table 5.1, we see that it is because of cost C_1 that RDQ performs worse than the PRAND. RDQ's benefit B_1 can increase only up to 100, whereas C_1 is introduced due to extra bandwidth usage and can exceed 100 depending on value of $n_{expected}$. In practice, the actual monetary cost associated with bandwidth usage may not be too significant. So, a smaller weight may be assigned to C_1 or C_1 may not even be included in the cost-benefit Analysis. In this case, the net benefit for RDQ would be $B_1 - C_2$, which then would be

always greater than the net benefit of PRAND, thereby causing RDQ to become a preferred choice over PRAND.

Another observation that can be made from Table 5.1 is that with increasing `port_id_bits`, cost C_2 decreases for both variants of RDQ. As `port_id_bits` increases, RDQ's PRAND behavior causes the attack probability p to approach *zero*. As seen from equation (5.11), cost C_2 will also approach *zero* in this case.

From Figs. 5.3-5.5, we observe that at lower guess rates, e.g., when `guess` = 10000, as `port_id_bits` increases, the net benefits of RDQ_1 become equal to those of RDQ_2. This again happens because with increasing `port_id_bits`, the attack probability p tends to *zero* and the probability q of obtaining the correct target IP address tends to *one*. As can be seen from Equation (5.8), this causes benefit B_1 to approach 100.

RDQ retries a query n times until a response is received that matches the untrusted response obtained by resolving the original client query. The expected number of such retries, $n_{expected}$, is computed using Equations (5.6) and (5.7). These equations show that as the attack probability p decreases, fewer retries are required. So, as p tends to *zero*, $n_{expected}$ tends to 1. It can be seen from Equation (5.10), as $n_{expected}$ tends to 1, cost C_1, the percentage increase in the CS's bandwidth usage because of RDQ, tends to 100. Moreover, as discussed in the previous paragraph, cost C_2 tends to *zero*. Therefore, the net benefits for both variants of RDQ approach *zero*.

5.5 Efficiency considerations

During our cost-benefit analysis of the countermeasures against the DNS cache poisoning and the DNS BAA, we evaluated a number of probabilistic reachability and reward properties. While model checking a property, PRISM loads the model in memory and creates a graph-based state-space representation of the model, which is then used for model checking. For a model to be useful in practice, the amount of memory needed by its graph-based state-space representation and the time required to evaluate a property should not be unreasonably high. In a PRISM model, only those parameters that appear in the guards for various actions affect the state-space size. Any parameters that are used as *rates* for various actions do not increase the size of a model's state-space. Also, in general, model checking reward properties takes more time than model checking probabilistic reachability properties. In our DNS cache poisoning model, we vary the `guess` rate and the number of `port_id_bits`. Both these parameters are used in expressions referring to rates; this helps limit the state-space size to 13 states and 16 transitions. The in-memory size of the model never exceeded 5.73 KB, whereas the model construction time never exceeded 0.005 sec. The time to model check the probabilistic reachability property was limited to 0.002 sec.

Chapter 6

Cost-Benefit Analysis for DNS Bandwidth Amplification Attack Countermeasures

We present the PRISM CTMC model for the DNS BAA and its five countermeasures, which include three primary countermeasures and two hybrid countermeasures. Using this model, we perform a cost-benefit analysis of the BAA countermeasures. We also show how this model can be used to analyze the impact of BAA on the DNSSec. (*Note:* Appendix B contains the fully documented model code)

6.1 CTMC model of DNS BAA

The basic CTMC model of the BAA consists of two primary modules.

- **Client Server (CS)**: CS is the victim server. It has a finite bandwidth, which is shared by legitimate DNS traffic and BAA traffic.
- **Net**: Net represents all DNS resolvers and clients that generate legitimate and attack traffic for the CS.

Model parameters: Since the CS is a DNS resolver, it handles both the legitimate DNS requests and the legitimate DNS responses. In order to let the CS process the legitimate DNS requests and the legitimate DNS responses in a uniform way, we define an abstraction called *legitimate DNS packet*. The size of a legitimate DNS packet is computed as the weighted average of the sizes of the legitimate DNS requests and the legitimate DNS responses that flow through the CS's network. The weights denote the frequencies with which the legitimate requests and the legitimate response are observed in the legitimate traffic. Since, DNS is a client-server system, one request generates one response. So, typically the legitimate DNS requests and the legitimate DNS responses appear with the same frequency. Then, we can compute the size of a legitimate DNS packet as the average of the size of a legitimate DNS request (60 bytes) and the size of a legitimate DNS response (512 bytes) [58]. A legitimate DNS packet size is, therefore, $\frac{60+512}{2} = 286$ bytes.

In BAA, the attacker sends many unwanted bogus DNS responses to the victim. As seen from section 2.4, each bogus DNS response is `AF` times larger than a *legitimate*

DNS request, i.e., AF = $\frac{size\ of\ bogus\ DNS\ response}{size\ of\ legitimate\ DNS\ request}$. However, the legitimate DNS traffic transmitted to the CS consists of both the legitimate DNS requests and the legitimate DNS responses, which are collectively represented using the abstraction *legitimate DNS packets*. So, we need to adapt the definition of AF to use the legitimate DNS packet size instead of the legitimate request size. The modified definition for the AF is, AF = $\frac{size\ of\ bogus\ DNS\ response}{size\ of\ legitimate\ DNS\ packet}$. The maximum size of a bogus DNS response can be 4380 bytes [58]. So, the AF is then computed as $\frac{4380\ \text{bytes}}{286\ \text{bytes}} = 15.31$.

We observe that because a bogus DNS response is AF times larger than a legitimate DNS packet, one bogus DNS response can be considered to be composed of AF legitimate DNS packets. So, the BAA, which introduces bogus DNS responses into the CS's network, can be considered to introduce additional legitimate DNS packets into the CS's network, where one bogus DNS response is equivalent to AF legitimate DNS packets. So, for a given number of bogus DNS responses, we can obtain the equivalent number of legitimate DNS packets by multiplying the number of bogus DNS responses with the AF. So, the number of bogus DNS responses and number of legitimate DNS packets can be measured using a single unit called *packets*. However, to do so, we need to ensure that we multiply the number of bogus DNS response by the AF.

The rate at which the legitimate DNS packets arrive at and flow out of the CS is called the R_l. As the consumption of CS's bandwidth chiefly depends on the attack strength, we set the R_l to a moderate value of 100 packets per second. In our model, the parameter zombies represents the number of zombies used in the BAA. Each zombie sends a fixed number of DNS responses to the CS per second. So, we can define the bogus_rate, i.e., the rate at which each zombie sends bogus DNS packets to the CS as follows.

$$\texttt{bogus\_rate} = number\ of\ bogus\ DNS\ reaponses\ sent\ per\ second \cdot \texttt{AF} \tag{6.1}$$

Since, it is the number of zombies that mainly influences the attack strength, we assume that each zombie sends a moderate number of 10 bogus DNS responses to the CS every second. So, the bogus_rate = $10 \cdot$ AF packets per second. Zombies further increase the arrival rate of bogus DNS packets. The net arrival rate for the bogus DNS packets, R_b, measured in packets per second is

$$R_b = \texttt{bogus\_rate} \cdot \texttt{zombies} \tag{6.2}$$

CS's bandwidth, BW, represents the finite capacity of the its network to simultaneously transmit the DNS traffic. Bandwidth is typically expressed in bits per second. A typical DNS server has approximately 1 Mbps [65] of dedicated bandwidth. Since, the legitimate DNS packet size is 286 bytes, we can say that 1 Mbps bandwidth can transmit $\frac{1\text{Mb}}{286} = 458$ packets per second. So, we set BW to 458 packets per second. Bandwidth can be modeled by using a finite-sized queue that represents the packets that have been transmitted to the CS and are waiting to be served. The growth and shrinkage of CS's available queue capacity models the way legitimate and bogus packets consume CS's bandwidth. The advantage of using a queue to represent the bandwidth is that we can take into consideration the effects of the rate at which the CS serves the incoming DNS packets, i.e., the serve_rate. The higher the serve_rate, the faster the CS processes queued packets, thereby freeing up the bandwidth at a faster pace. A typical value for serve_rate is 12666 packets per

Description	Parameter	Value	Units
Amplification factor	AF	15.31	
Legitimate DNS packet rate	R_l	100	packets per second
Number of zombies	zombies		
Rate at which each zombie sends the bogus DNS packets	bogus_rate	$10 \cdot$ AF	packets per second
Net arrival rate for bogus DNS packets	R_b	bogus_rate $\cdot$ zombies	packets per second
CS's bandwidth	BW	458	packets per second
CS's serve rate	serve_rate	12666	packets per second

Table 6.1: DNS BAA model parameters

second [1]. When a queued packet is served, the available queue capacity is incremented by 1. Table 6.1 summarizes the model parameters and constants.

We now describe the three primary actions in our BAA model.

- **Receive Legit Packet:** A legitimate DNS packet is received by the CS and a position in CS's queue is occupied. This action is defined in modules **CS** and **Net** and occurs at the rate R_l for the Nofix case, i.e., when no countermeasure is applied.
- **Receive Bogus Packet:** This action is defined in modules **CS** and **Net** and represents the receipt of bogus packets by the CS and the corresponding reduction of the available queue capacity. For the Nofix case, this action occurs at the rate of R_b.
- **Client Request:** This action is defined in modules **CS** and **Net** and represents the dispatching of a client request to the CS with rate R_l. This action can be executed if the queue capacity is available. After having been executed once, this action is permanently disabled. It is used as a handle for computing the attack probability.

The CTMC model of the BAA can be extended to model the three basic countermeasures as follows. Table 6.2 summarizes the countermeasure parameters.

- **Modeling FTR:** FTR is implemented by augmenting the basic CTMC BAA model with a new module called the **Filter**. Filter is configured using two parameters: detection_fraction (df) and false_positive_fraction (fpf). The df is the fraction of attack traffic identified and filtered, whereas the fpf is the fraction of legitimate traffic incorrectly identified as bogus. Studies of different filtering algorithms show that on average FTR has a high df of 0.9 and fpf as low as 0.1. FTR decreases the effective rate of the action **Receive Legit Packet** to $R_l \cdot (1-$fpf$)$ and the effective rate of the action **Receive Bogus Packet** to $R_b \cdot (1-$df$)$.

Description	Parameter
Fraction of bogus DNS packets correctly filtered	`df`
Fraction of legitimate DNS packets incorrectly filtered	`fpf`
Fraction of bogus and legitimate DNS packets randomly dropped	`rdf`
Number of times the legitimate DNS packets are resent	`retries`

Table 6.2: DNS BAA countermeasure parameters

- **Modeling RND:** RND is implemented by a new module called the **RandomDropper**. It is controlled by the parameter `random_drop_fraction (rdf)`, the fraction of incoming legitimate and bogus packets randomly dropped. RND reduces the effective rate of the action **Receive Legit Packet** to $R_l \cdot (1-\texttt{rdf})$ and the effective rate of the action **Receive Bogus Packet** to $R_b \cdot (1-\texttt{rdf})$.

- **Modeling AGR:** AGR is implemented by adding a new parameter, the `retries` to the basic CTMC BAA model. The `retries` represent the number of times the legitimate DNS packets are resent to increase the share of legitimate traffic. AGR increases the effective rate of the action **Receive Legit Packet** to $R_l \cdot 2^{\texttt{retries}}$.

Now, we explain how to use the basic CTMC BAA model for modeling the two hybrid countermeasures.

- **Modeling RDR:** RDR, the combination of RND and AGR, is implemented using the module **RandomDropper** along with the parameter `retries`. RDR is thus controlled by `rdf` and `retries`. It increases the effective rate of the action **Receive Legit Packet** to $R_l \cdot (1-\texttt{rdf}) \cdot 2^{\texttt{retries}}$ and decreases the effective rate of the action **Receive Bogus Packet** to $R_b \cdot (1-\texttt{rdf})$.

- **Modeling AGF:** AGF is combination of AGR and FTR. It is implemented using the module **Filter** and the parameter `retries`. AGF is controlled by `retries`, `df`, and `fpf`. It increases the effective rate of the action **Receive Legit Packet** to $R_l \cdot (1-\texttt{fpf}) \cdot 2^{\texttt{retries}}$ and decreases the effective rate of the action **Receive Bogus Packet** to $R_b \cdot (1-\texttt{df})$.

Modeling the DNSSec BAA: As we showed in [20], the PRISM model for the DNSSec BAA can be obtained by assigning proper values to `BW` and `AF`. For DNSSec, `BW` = 112 and `AF` = 16.32. The DNSSec has a high AF and its response size can be much larger than a DNS response. So, fewer DNSSec packets can be accommodated in a bandwidth of a given capacity.

6.2 Benefit and cost metrics

During a BAA, the CS's bandwidth is consumed by a large number of unwanted amplified DNS responses. So, the legitimate client requests may end up not getting serviced. The CS has to, however, ensure that legitimate client requests are still serviced. The CS, therefore, needs to ensure that the *attack probability*, i.e., the probability that a legitimate client request is eventually not satisfied, is set to *zero*. The observed attack probability,

p, is computed using the CSL formula `P=? [F DenialOfService]` with predicate `DenialOfService` becoming true, when the **Client Request** action is not completed by the CS under attack. To achieve the *zero* attack probability for its clients, the CS applies the countermeasures. It is possible to ensure the *zero* attack probability by adjusting countermeasure parameters such as `rdf` for RND and `retries` for AGR.

Once the CS ensures that is is able to secure the *zero* attack probability for its clients, it can try to ensure that its own capabilities are efficiently utilized. E.g., the CS may try to ensure that the majority of the packets that it processes are the legitimate packets. The CS may also want to process as few packets as possible. It would also like to ensure that fewer legitimate packets are dropped because of the false-positive action of the countermeasures such as FTR and RND. We define two benefit metrics and one cost metric to measure how efficiently a countermeasure can help the CS to better utilize its capabilities. All benefit and cost metrics are computed as percentage values. Benefit B_1 is the percentage of legitimate packets in total packets processed. A good countermeasure offers high B_1. Benefit B_2 is the percentage time spent in the states where the bandwidth is available. A high value of B_2 ensures that a countermeasure keeps the victim's bandwidth free, signifying that the victim receives as few packets as possible. Cost C_1 represents the false positive-based cost associated with FTR, RND, RDR, and AGF. C_1 is defined as the percentage of the legitimate packets dropped.

For computing benefit and cost metrics, we define three reward properties P_1, P_2 and P_3 of the form `R{"<reward definition>"}=? [C<=t]`, that evaluate the accumulated quantities in time t for rewards R_1, R_2, R_3 as follows. The transition reward R_1 assigns a unit yield to the actions **Receive Legit Packet** and **Client Request**. So, property P_1 counts the total number of legitimate packets received, say PK_1. R_2 is another transition reward with unit yield attached to the action **Receive Bogus Packet**. The property P_2 counts the total number of bogus packets received, say PK_2. Then,

$$B_1 = \frac{PK_1}{PK_1 + PK_2} \cdot 100 \tag{6.3}$$

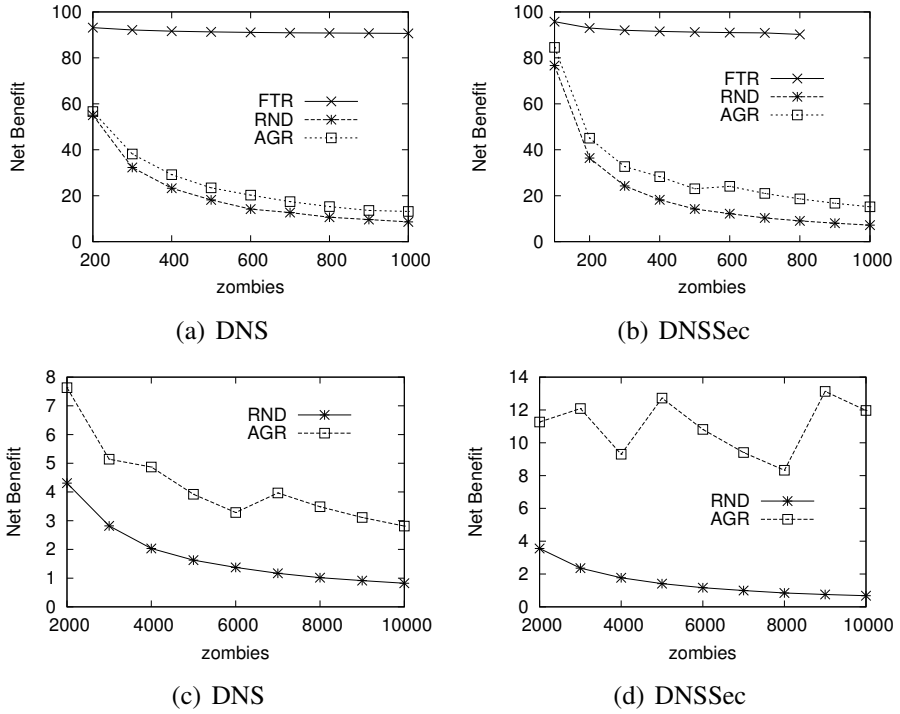
Reward R_3 is a state reward that assigns a unit yield per unit of time spent in the states where the victim's bandwidth has not been exhausted. Therefore, property P_3 yields the total time T_1 spent in states where bandwidth is available. Then,

$$B_2 = \frac{T_1}{t} \cdot 100 \tag{6.4}$$

where t is the time duration for which the countermeasure effects are evaluated, which is same as the time-bound used for evaluating P_3.

Both FTR and AGF drop `fpf` fraction of the incoming legitimate traffic. Similarly, RND and RDR both drop `rdf` fraction of the incoming traffic, including bogus and legitimate traffic. AGR does not have any false positives. Therefore,

$$C_1 = \begin{cases} \texttt{fpf} \cdot 100, & \text{for FTR and AGF} \\ \texttt{rdf} \cdot 100, & \text{for RND and RDR} \\ 0, & \text{for AGR} \end{cases} \tag{6.5}$$



(a) DNS (b) DNSSec

(c) DNS (d) DNSSec

Figure 6.1: Net benefits for FTR, RND, and AGR

Finally the net benefit is computed as

$$net\ benefit = B_1 + B_2 - C_1 \tag{6.6}$$

6.3 Experimental results

The cost-benefit analysis of the DNS BAA countermeasures is an example of the cost-benefit analysis "with model repair". This approach was selected, since all BAA countermeasures have different control parameters, e.g., the `df` and the `fpf` for FTR, the `rdf` for RDR, and the `retries` for AGR. So, the control parameters for the BAA countermeasures cannot be used as the basis for comparing the countermeasure performance. However, we can use a "desired attack probability" as the basis for comparing the countermeasure performance. The cost-benefit analysis is, therefore, performed in two phases. In the first phase, we use model repair to determine the *minimum* values of the countermeasure parameters that produce the *zero* attack probability for different attack settings, i.e., for different number of `zombies`. If during the first phase, we find that the countermeasure is indeed able to achieve the *zero* attack probability, then we perform the second phase. For RND, AGR, RDR, and AGF it is theoretically possible to adjust the `rdf` and the `retries` to achieve the *zero* attack probability. However, for FTR, the `df` and the `fpf` are fixed, so it is not always possible to achieve the *zero* attack probability and the observed attack probability, p, can be more than *zero*. Therefore, the second phase is always performed for RND, AGR, RDR, and AGF. However, the second phase for FTR is performed only if the FTR can achieve the *zero* attack probability. In the second phase, we use the countermeasure parameter values identified in the first phase to compute the costs and the benefits, and subsequently the net benefits. We can then select the countermeasure

that offers the highest net benefit for a given attack setting.

In [20], we analyzed the performance of FTR, RND, and AGR. We reproduce those experimental results here. Since effects of the BAA have to be mitigated in a small time interval, the time-bound t referred in the equation (6.4) is selected, such that it allows the model dynamics to reasonably evolve, while at the same time prevents it from attaining steady-state (where countermeasure's effects are independent of the available bandwidth). By experimentation we found $t = 0.1$.

From Figs. 6.1(a)-6.1(d) we observe, that for all countermeasures, the net benefit decreases with increasing number of `zombies`. This shows that the protection offered by a countermeasure weakens as the attack strength rises. From Figs. 6.1(a) and 6.1(c), we see that for DNS, FTR is the most cost-effective countermeasure when `zombies` ≤ 1000. Thereafter, AGR offers the highest net benefit. The FTR has a constant high `df` with low `fpf`. So, FTR, in general, offers a superior performance. However, when the number of `zombies` exceeds a certain threshold, e.g., 1000 for DNS, then FTR cannot achieve the *zero* attack probability. Therefore, the second phase of the cost-benefit analysis is not performed for FTR when the number of `zombies` exceed 1000. This can be clearly seen from Fig. 6.1(c), which shows the net benefits for only RND and AGR. A similar behavior is observed for DNSSec from Figs. 6.1(b) and 6.1(d), where FTR is the best countermeasure when `zombies` ≤ 800. Thereafter, AGR becomes the countermeasure offering the highest net benefit. AGR has no false positives and it proactively increases the proportion of the legitimate DNS traffic. So, it can cope up with high number of `zombies`.

We can thus conjecture, that the combination of FTR and AGR, i.e., the AGF might offer the best performance. It would be also interesting to study how the RDR, i.e., the combination of RND and AGR, behaves. We can expect RDR to perform better than RND. The combination of FTR and RND is less interesting, since both FTR and RND are based on the same principle of filtering out the bogus traffic. Therefore, we extended our BAA model to analyze AGF and RDR.

Determining countermeasure parameters for AGF and RDR:

- **RDR:** Since RDR is a combination of RND and AGR, we determine `rdf` and `retries`. For a given number of `zombies`, we know the `rdf` required by RND and the `retries` required by AGR to achieve the *zero* attack probability. E.g., when `zombies` = 500, then for RND, `rdf` = 0.82 and for AGR, `retries` = 4. Clearly for RDR, $0 <$ `rdf` < 0.82 and $0 <$ `retries` < 4. So, we set `retries` to 1, 2, and 3 and for each retry, identify the minimum `rdf` required to produce the *zero* attack probability. For each set of (`retries`, `rdf`) we compute the net benefit. We choose the (`retries`, `rdf`) that offers the highest net benefit for a particular number of `zombies`.

- **AGF:** Since AGF is a combination of FTR and AGR, we need to determine `df`, `fpf`, and `retries`. The `df` and the `fpf` are constant with `df` = 0.9 and `fpf` = 0.1. We initialize `retries` to *zero* and keep it constant as long as the filtering mechanism of AGF can maintain the *zero* attack probability. Thereafter, we identify the lowest possible number of `retries` that in combination with filtering can reduce the attack probability to *zero*. E.g., for DNS we observe, that the FTR can provide the *zero* attack probability when `zombies` ≤ 1000. For `zombies` > 1000,

zombies	FTR		RND	AGR	RDR		AGF		
	df	fpf	rdf	retries	rdf	retries	df	fpf	retries
200	0.9	0.1	0.44	2	0.4	1	0.9	0.1	0
500	0.9	0.1	0.82	4	0.79	2	0.9	0.1	0
800	0.9	0.1	0.89	5	0.87	3	0.9	0.1	0
2000			0.96	7	0.95	4	0.9	0.1	3
5000			0.98	8	0.98	5	0.9	0.1	5
8000			0.99	9	0.99	6	0.9	0.1	6

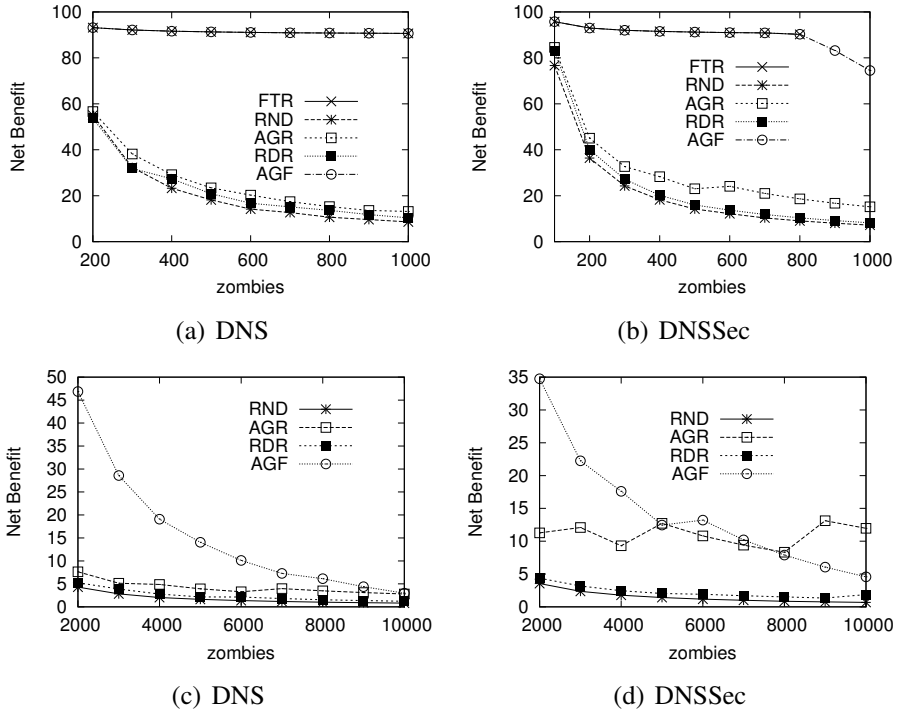
Table 6.3: DNS BAA countermeasure parameters required to achieve zero *attack probability*

zombies	FTR			RND			AGR		RDR			AGF		
	B_1	B_2	C_1	B_1	B_2	C_1	B_1	B_2	B_1	B_2	C_1	B_1	B_2	C_1
200	3.2	100	10	0.4	98.7	44	0.9	55.8	0.5	93.1	40	3.2	100	10
500	1.3	100	10	0.2	100	82	1.2	22.3	0.4	99.9	79.5	1.3	100	10
800	0.8	100	10	0.1	100	89.5	1.4	13.9	0.4	99.9	86.7	0.8	100	10
2000				0.1	100	95.8	2.1	5.5	0.3	99.9	95.1	1.4	55.5	10
5000				0.03	100	98.4	1.7	2.2	0.3	99.9	98.1	2	22.1	10
8000				0.02	100	99	2.1	1.4	0.3	99.9	98.8	2.4	13.7	10

Table 6.4: DNS BAA countermeasures costs and benefits

we vary `retries` to achieve *zero* attack probability and record the least number of `retries` for which the *zero* attack probability is observed.

We now present the results of the cost-benefit analysis for RDR and AGF. Table 6.3 shows the countermeasure parameter values necessary to achieve the *zero* attack probability for the DNS. The empty cells in the table indicate that FTR's `df` of 0.9 and `fpf` of 0.1 are insufficient to achieve the *zero* attack probability for the given number of zombies. For DNS, Figs. 6.2(a)-6.2(d) show the net benefits offered by FTR, RND, AGR, RDR, and AGF. These net benefits have been computed using countermeasure parameter values from Table 6.3. Table 6.4 shows the costs and benefits observed for various BAA countermeasures. From Figs. 6.2(a)-6.2(d), we observe that AGF offers the highest net benefit. For `zombies` ≤ 1000, both AGF and FTR offer identical net benefits since, as seen from Table 6.3, when `zombies` ≤ 1000, for AGF, the `retries` are set to *zero*. When `zombies` > 1000, AGF can use its aggressive retries to ensure that the attack probability still remains *zero*. Also, AGF's capability of filtering bogus packets with low false positives ensures that the benefit B_2 remains high in presence of aggressive retries. This



(a) DNS (b) DNSSec

(c) DNS (d) DNSSec

Figure 6.2: Net benefits for FTR, RND, AGR, RDR, and AGF

lets the AGF to have the highest net benefit.

RDR, on the other hand, offers higher net benefit than RND, but lower net benefit than AGR. RDR's aggressive retries increase benefit B_1 and marginally decreases cost C_1, thereby causing RDR's net benefit to be greater than the net benefit of the RND. However, RDR's random drop action affects the redundant legitimate traffic introduced by the aggressive retries. This causes RDR's B_1 to be strictly less than the B_1 of the AGR. Therefore, RDR's net benefit is always less than AGR's net benefit.

For DNSSec, the relative performance of BAA countermeasures is similar to their performance for the DNS. However, DNSSec, being more susceptible to the BAA, needs higher values of the countermeasure parameters for achieving the *zero* attack probability. So, as seen from Figs. 6.2(a)-6.2(d), generally all countermeasures provide less net benefit for DNSSec than the net benefit they provide for DNS. However, we see that when $\texttt{zombies} > 500$, AGR is more beneficial for DNSSec than it is for DNS. AGR works by resending the legitimate packets at a doubled rate during each retry. The average size of a legitimate DNSSec packet is 1163.5 bytes as compared to an average legitimate DNS packet size of 286 bytes. So, AGR increases volume of legitimate traffic for DNSSec far more rapidly as compared to DNS, which rapidly increases B_1, hence increasing the net benefit.

6.4 Observations

As seen from section 6.3, the cost-benefit analysis not only helps us to identify the best performing countermeasure, but also lets us develop new countermeasures. Originally, we modeled the countermeasures described in the literature, i.e., FTR, RND, and AGR. However, the cost-benefit analysis allowed us to design and analyze interesting combinations

of these individual countermeasures. This led us to AGF, the countermeasure with the highest net benefit.

6.5 Efficiency considerations

In the DNS BAA model, the parameter `BW` appears in a guard and therefore, affects the state-space size. The state-space size increases with the value of the `BW`. Other parameters such as the number of `zombies`, `df`, `fpf`, `rdf`, `retries`, and `AF` are used in expressions of rates, so they do not affect model's state-space size. For DNS the `BW` = 458, whereas for DNSSec the `BW` = 112. So, the size of the DNSSec BAA model is, in fact, smaller than the size of the DNS BAA model. For DNS BAA model, the state-space consists of 918 states and 2292 transitions. The in-memory size of the model never exceeded 1.73 KB and the model construction time was 0.087 sec. The maximum times to model check probabilistic reachability property and the reward properties were 1.093 sec and 3.99 sec respectively. For DNSSec BAA model, the state-space contains 226 states and 512 transitions. The maximum in-memory size of the model was 1.39 KB and the model construction time was 0.027 sec. The time to model check the probabilistic reachability property and the reward properties never exceeded 0.246 sec and 1.044 sec respectively.

Chapter 7

Stochastic Game-Based Modeling with PRISM

7.1 From MDPs to Stochastic Games

As discussed in Chapter 3, a model in PRISM is constructed as the parallel composition of its modules. A module is described by a collection of commands or actions, each of which comprises a guard and one or more updates:

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

The guard $\mathtt{g}$ is a predicate over model variables, whereas each update $\mathtt{u}_i$ describes by assigning new values to the model variables, a transition that the module can make. For DTMCs and MDPs, λ_i is the *probability* with which update $\mathtt{u}_i$ takes place while for CTMCs, λ_i represents the *transition rate* associated with update $\mathtt{u}_i$. DTMCs and CTMCs both allow only stochastic transitions, so in a DTMC or a CTMC model, at any given state only one command can be active. However, MDPs allow both stochastic and nondeterministic behaviors. Therefore, in an MDP, multiple commands can remain enabled in a given state and one of those commands is nondeterministically selected.

A stochastic game is played with one or more players. At every state s of the game, a player has several possible moves or actions to choose from. The game begins with some initial state. Each player P_i then chooses an action a_i from the set of actions $A_i(s)$ *available* for player P_i in state s. The player P_i gets a *payoff*[1] $r(s, a_i)$ that depends on the current state and the chosen action. Once each player P_i selects an action $a_i \in A_i(s)$, the next state is chosen according to the probability distributions $p(s, a_i)$. The next state, therefore, depends on the current state s and the probability distributions $p(s, a_i)$. This process is repeated in the new state and the game can continue for a finite or infinite number of steps. The total payoff is the sum of the payoffs obtained in each step [44].

We know that a PRISM MDP model is composed of a number of modules where each module consists of one or more commands. At any given state, one or more commands are enabled, thereby indicating the set of actions *available* in that state. At each state, one of the available action is nondeterministically chosen and the model moves into the next state which is determined by the current state and the probability distribution associated

[1] Payoff represents desirability of the game's outcome for each player [52].

with the selected action. This is accompanied by accumulation of the reward associated with the selected action. The same procedure is repeated at each state of the model.

So, we observe that in both MDPs and stochastic games, at any given state, the players involved choose an action. Each player gets some payoff determined by the action chosen. The model (or the game) then proceeds to the next state, which is determined by the current state and the probability distribution associated with the chosen action. The same procedure is repeated in the new state. As we can see, there is a close similarity between an MDP and a stochastic game. In fact, MDPs are stochastic games with a single player [44].

7.2 PRISM-games

PRISM always had support for model checking MDPs [23]. PRISM-games extended PRISM's MDP model checking capabilities to support *Stochastic Multi-player Games (SMG)*, in order to analyze systems with both *probabilistic* and *competitive* behaviors [15]. PRISM-games supports analysis of a subset of stochastic games, viz., the two-player, turn-based, zero-sum stochastic games. Such games are played between only two players and at any given stage of the game only one player is allowed to make a move. Moreover, these games are zero-sum games, meaning that the gain of one player is exactly balanced by the loss of the other player [63]. Turn-based games can be used to model situations where multiple components execute under the control of a central scheduler and each component nondeterministically chooses an action. Zero-sum games typically arise in situations where two players are competing for a resource such as bandwidth. So, one player's gain is other player's loss and the total of all gains and losses equals *zero*.

PRISM-games supports rPATL, an extension of the Probabilistic Alternating-time Temporal Logic. rPATL can be used to specify quantitative properties for SMGs. Using rPATL, we can write properties, which for a coalition of players, can identify a strategy such that either an expected probability of an event or an expected value of the accumulated reward is maximized or minimized.

Since PRISM-games is a fairly new tool (released in 2012), only a few case studies featuring it are available. Four case studies featuring PRISM-games are available:

- Microgrid Demand-Side Management (MDSM) [15]
- Collective Decision Making for Sensor Networks (CDMSN) [15]
- Futures Market Investor (FMI) [42]
- Team Formation Protocol (TFP) [17]

In the MDSM case study, PRISM-games was used to design a protocol that would incentivize households participating in a microgrid to adhere to the policy of fair usage while submitting electrical loads to the central distribution manager. Before submitting a load, each household checks if the cost is within an agreed limit. If yes, then the household executes a job, else it starts the job with a pre-negotiated probability. This policy reduces the peak demand and the total cost of energy as long as all households adhere to this policy. However, it is always possible for an unscrupulous household to deviate from this policy and execute a load even if its cost exceeds the agreed limit. The original MDSM protocol

did not have any provision for punishing the offender, however, the PRISM-games model of the MDSM helped to devise a strategy that would deter a household from deviating from the MDSM protocol. By using PRISM-games, it was found that the distribution manager can punish a deviant household by canceling one job per time step if the cost exceeds the limit. Because of this, a household that constantly abuses the system can find its jobs being canceled, which would force it to adhere to the MDSM protocol.

In the CDMSN case study, PRISM-games was used to discover the optimal size for a coalition of sensors to ensure that the sensor network can quickly recover from failure of one or more sensors, thus, improving its robustness. The optimal coalition size also minimizes the time taken to establish consensus among the sensors to decide upon a target to which the information is to be transmitted. The FMI case study analyzes the futures market scenario where the investor tries to maximize his return against the futures market which in turn tries to minimize the investor's return. In the TFP case study, PRISM-games was used to find the optimal size for the coalition of agents which cooperate to achieve a goal irrespective of any strategies employed by the agents hostile to them.

7.2.1 Structure of a PRISM SMG

A PRISM SMG model consists of i modules, M_1, M_2, ..., M_i. A module is used to group together related commands. E.g., in our model, the defender module consists of commands that enable several BAA countermeasures. We can define players P_1, P_2, ..., P_j, such that each player comprises of one or more modules. A module can be part of multiple players as long as we ensure that each command in a module is assigned to exactly one player. E.g., if a module M consists of commands C_1, C_2, C_3, and C_4, then commands C_1 and C_2 could be part of player P_1 and commands C_3 and C_4 could be part of player P_2. The `player ...endplayer` construct is used to signify a player and all commands under its control, e.g., we can define player P_1 as,

```
player P1
    C1, C2
endplayer
```

A PRISM SMG model can, thus, contain multiple players. However, PRISM-games supports only two-player games. To satisfy the two-player constraint while model checking a property, we are required to divide all players in a SMG into two groups or *coalitions* with each group representing one player. These two groups of players act as *adversaries* of each other. E.g., the coalition of players P_1 and P_2 is specified as $<< \mathtt{P_1}, \mathtt{P_2} >>$. All remaining players form the other coalition.

7.2.2 Cumulative rewards in PRISM-games rPATL and optimal strategies

PRISM allows us to specify and analyze properties based on *costs* and *rewards*, which help us to evaluate interesting quantitative properties about a model's behavior. E.g., reward-based properties can be used to compute properties such as "expected time to reach a desired state" or "expected power consumption. In a game-based model, we can use reward-based properties to define the players' payoffs. So, it is important to understand

how PRISM evaluates the reward-based properties. For simplicity, we explain evaluation of reward-based properties for the DTMCs. Consider a DTMC $D = (S, \overline{S}, P, L)$, where S is a finite set of states, $\overline{S} \in S$ is the initial state, $P : S \times S \to [0, 1]$ is the transition probability matrix such that $\sum_{s' \in S} P(s, s') = 1$ for all $s \in S$, and $L : S \to 2^{AP}$ is a labeling function which assigns to each state $s \in S$ the set $L(s)$ of atomic propositions that are valid in the state. PRISM supports two kinds of rewards: state rewards ($\rho : S \to \mathbb{R}_{\geq 0}$) and transition rewards ($\iota : S \times S \to \mathbb{R}_{\geq 0}$). PRISM's reward-based properties allow us to compute the *expected* accumulated values of a reward until a target set $T \subseteq S$ is reached. $ExpReach(s, T)$, the expected reward accumulated starting from state s until the target states $T \subseteq S$ are reached, is defined using following linear equation system [38]:

$$ExpReach(s, T) = \begin{cases} 0 & \text{if } s \in T \\ \infty & \text{if } ProbReach(s, T) \leq 1 \\ \rho(s) + \sum_{s' \in S} P(s, s') \cdot (\iota(s, s') + ExpReach(s', T)) & \text{otherwise} \end{cases} \tag{7.1}$$

$ProbReach(s, T)$ is the probability that a path starting in state s would reach a state in the target set $T \subseteq S$. It is defined as following linear equation system [8]:

$$ProbReach(s, T) = \begin{cases} 1 & \text{if } s \in T \\ 0 & \text{if } T \text{ is not reachable from } s \\ \sum_{s' \in S} P(s, s') \cdot ProbReach(s', T) & \text{otherwise} \end{cases} \tag{7.2}$$

From equation (7.1), we see that if we use only the state rewards and if while evaluating $ExpReach(s, T)$ the cycles in the state transition graph cause states s' to be repeatedly visited, then $ExpReach(s, T)$ would start to stabilize.

The rPATL extends reward-based property evaluation approach from equation (7.1) to SMGs that contain n players, $\mathtt{P_1}$, $\mathtt{P_2}$, ..., $\mathtt{P_n}$. Let us consider a reward-based rPATL property in PRISM-games:

$$<< \mathtt{P_1}, \mathtt{P_2}, \ldots, \mathtt{P_k} >> \mathtt{Rmax/min}\{\text{“}\mathtt{rewardDef}\text{”}\} =? [\mathtt{F}\ \phi] \tag{7.3}$$

This property asks the PRISM-games to generate an optimal strategy for the coalition C_1 of players containing players $\mathtt{P_1}$, $\mathtt{P_2}$, ..., $\mathtt{P_k}$ where $(k < n)$. Such optimal strategy either maximizes or minimizes the expected accumulated value of reward “`rewardDef`” until some state from the target set of states that satisfy the state formula ϕ is eventually reached. Since the game is also a zero-sum game, PRISM-games simultaneously generates the optimal strategy for the adversarial coalition C_2 of players $\mathtt{P_{k+1}}$, $\mathtt{P_{k+2}}$, ..., $\mathtt{P_n}$, such that it reduces reward obtained by coalition C_1. PRISM-games returns both the generated optimal strategy and the expected accumulated reward. In case the target set of states is not reached, the accumulated reward value is set to infinity.

Optimal strategies generated by PRISM-games and Nash equilibrium: Typically, the goal of a game-theoretic analysis is to determine if there exists a *Nash equilibrium* for a game. A Nash equilibrium represents a set of strategies for all players in a game, where

no player can hope to better its payoff by unilaterally switching to another strategy [62]. In game-based models of security attacks, Nash equilibria represent optimal strategies for defender and attacker, by virtue of which, the defender can minimize the impact of the attack and the attacker can cause the maximum damage. Therefore, it is interesting to study the relationship between optimal strategies generated by PRISM-games and Nash equilibrium. Let us assume a two-player zero-sum game with players $\texttt{P}_1$ and $\texttt{P}_2$. Let us consider a property $prop_1$: $<<\texttt{P}_1>>\texttt{Rmax}\{\text{“}\texttt{reward}_1\text{”}\} =?\,[\texttt{F}\ \phi_1]$. This causes the PRISM-games to generate a strategy for $\texttt{P}_1$, using which $\texttt{P}_1$ can guarantee that the expected value of reward “$\texttt{reward}_1$” accumulated before reaching states satisfying ϕ_1 is maximized. This means that $\texttt{P}_1$ has identified a strategy that would allow it to maximize the expected accumulated value of reward “$\texttt{reward}_1$” in face of the maximum opposition from $\texttt{P}_2$. So, $\texttt{P}_1$ has no incentive to switch to any other strategy, since it is bound to reduce the expected accumulated value of reward “$\texttt{reward}_1$” (or the payoff). Similarly, in this process, $\texttt{P}_2$ has identified a strategy that would minimize $\texttt{P}_1$'s payoff. Since the game is a zero-sum game, any reduction in $\texttt{P}_1$'s payoff is a gain for $\texttt{P}_2$. So, $\texttt{P}_2$ too has maximized its own payoff and it has no incentive to switch to any other strategy. Therefore, we can infer that an optimal strategy generated by PRISM-games is indeed a Nash equilibrium.

Analyzing the nature of an optimal countermeasure strategy: Once PRISM-games generates an optimal strategy, we can explore it interactively using the PRISM simulator [16]. We enter the desired model parameter values and explore the model by executing one move at a time. At every state explored, the simulator highlights the move suggested by the optimal strategy. By executing these moves, we can continue our exploration of the model, and at each state we can record the optimal action chosen.

Chapter 8

Game-Based Model of the DNS BAA

8.1 Two-player, turn-based stochastic game for the DNS BAA

The DNS BAA game is played between two players, the attacker and the defender. The attacker tries to flood victim's bandwidth with unwanted amplified DNS responses while the defender tries to mitigate the attack by employing various countermeasures. Fig. 8.1 shows the schematic diagram of our modeling approach for the DNS BAA. Fig. 8.1 shows various entities involved in the DNS BAA, such as the attacker, the legitimate clients, the defender, and the victim DNS resolver. Through the remainder of this section, we would discuss how these entities are modeled in our two-player, turn-based stochastic game for the DNS BAA. (*Note:* Appendix C contains the fully documented model code)

8.1.1 Modules and players

The stochastic game for the DNS BAA developed using the PRISM-games model checker consists of two modules corresponding to the two players. We also have a third module to represent the victim server. So, the stochastic model for the DNS BAA consists of following three modules. These modules have also been shown in Fig. 8.1.

- **Attacker (AS):** Nondeterministically chooses the number of zombies to launch the BAA. Attacker can use up to 100 zombies to launch the BAA. Attacker can disable the attack by setting the number of zombies to *zero*. Moreover, we assume that the attacker can attack continuously for at most `MAX_SUCC_ATTACKS` seconds. Then, the attacker must wait for `ATTACKER_LATENCY` seconds before it can attack again.

  It should be noted that as seen from Fig. 2.3, the victim receives amplified bogus DNS responses from the open DNS resolvers that resolve the spoofed queries generated by zombies. However, because the open DNS resolvers simply *reflect* the bogus DNS responses to the victim, while modeling the attack it is reasonable to assume that the bogus traffic originates directly from the zombies. So, we exclude open DNS resolvers from our model. Moreover, since each zombie sends bogus DNS responses at a fixed rate, we could assume that the bogus traffic originates from the attacker (Step 1.1 from Fig. 8.1). The rate at which the bogus traffic arrives can, therefore, be computed by multiplying the rate at which each zombie sends bogus DNS responses to the victim with the number of zombies.

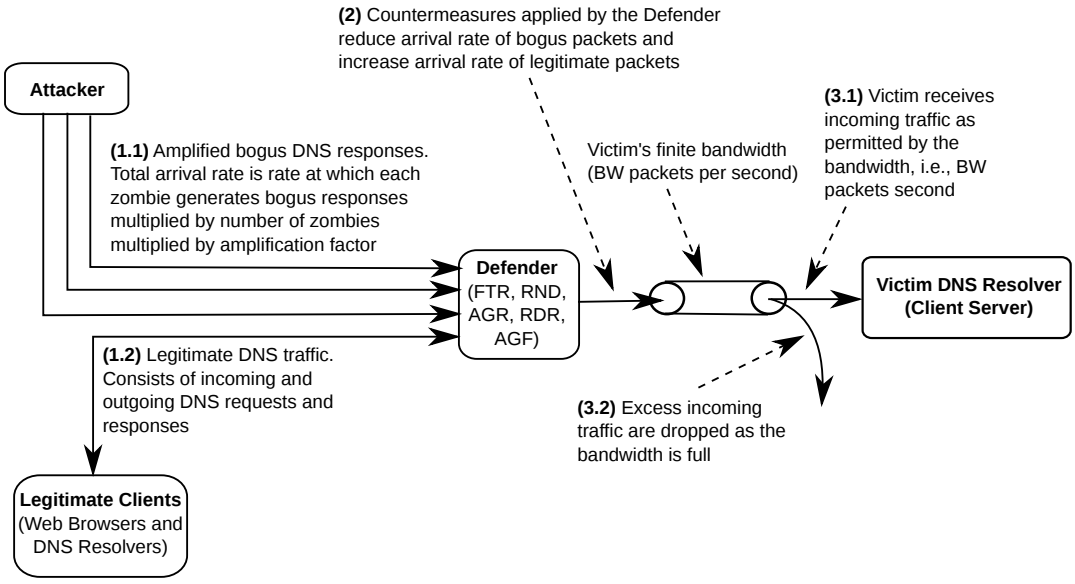


Figure 8.1: Modules and players in DNS BAA stochastic game

- **Defender (DFD):** Nondeterministically determines whether to disable countermeasures (Nofix) or apply one of the five BAA countermeasures. If defender opts to enable the countermeasures, then it may nondeterministically select FTR, AGR, RND, RDR, or AGF (Step 2 from Fig. 8.1).

- **Client Server (CS):** CS is the victim server. It is a DNS resolver. CS has a finite network bandwidth, which is shared by the legitimate DNS traffic (Step 1.2 from Fig. 8.1) and the BAA traffic (Step 1.1 from Fig. 8.1). CS's bandwidth is `BW` packets per second. If the packets arrive at a rate r, which is higher than the available bandwidth, then the excess packets, `BW`-r are dropped per second (Step 3.2 from Fig. 8.1).

Our model does consist of three players, however, PRISM-games support only two-player games. So, we divide these three players into two groups or *coalitions*. The first coalition contains the AS and the second coalition contains the DFD and the CS. Each group represents one player and the two groups act as adversaries of each other.

Model constants and parameters: The model constants and parameters as well as the countermeasure parameters are same as those described in section 6.1. However, we introduce a new parameter, called `maxTime` that represents the maximum time for which the model is allowed to execute. Note that we were not required to model time explicitly in the CTMC model presented in chapter 6, as CTMCs model the time implicitly. Also, in the DNS BAA SMG, we no longer model bandwidth using a queue. Instead, we model the bandwidth simply as a rate. This decision was necessitated by a need to limit the DNS BAA SMG's state-space size. Modeling the time explicitly increases the state-space. If the bandwidth is modeled using a queue, then the state-space size becomes too large for the model checker to handle. Moreover, to study the effects of the BAA it is sufficient to ensure that if the packet arrival rate is less than bandwidth, then all packets are *received*, and that if the packet arrival rate is greater than bandwidth, then the excess packets are

dropped. If CS's `serve_rate` is sufficiently high, then all packets received per second would be processed instantaneously. So, we can remove the parameter `serve_rate` as well.

8.1.2 Sequence of moves

PRISM-games requires that the players in an SMG play the game in a turn-based manner. So, we need to ensure that in any given state of the game, only one player is able to make a move. This is achieved by introducing a *scheduling* variable named `sched`. If `sched` = 0, then CS is enabled, if `sched` = 1, then DFD is enabled, and if `sched` = 2, then AS is enabled. This leads to following sequence of moves:

1. Initially, AS is enabled (`sched` = 2). Time is set to 0
2. AS nondeterministically chooses the number of `zombies`
3. AS enables DFD (`sched` = 1)
4. DFD nondeterministically chooses one of the five countermeasures or decides to disable the countermeasures
5. DFD enables CS (`sched` = 0)
6. CS receives legitimate and bogus packets with probabilities that reflect the selected countermeasure, the arrival rates of the legitimate and bogus DNS packets, and the CS's bandwidth.
7. CS enables AS (`sched` = 2) and increments time by 1
8. Goto step 2

This sequence of moves is repeated for the desired duration of the experiment.

8.1.3 DNS BAA SMG

Fig. 8.2 shows the SMG for the DNS BAA. In an SMG, each model state is controlled by only one player. The player P_i controlling a state S_j is represented as $P_i : S_j$, e.g., $AS : S_{init}$ means that only AS's actions are enabled in S_{init} state. At every state, the controlling player nondeterministically chooses one of the available actions. In Fig. 8.2, the label on each transition denotes the model action a, along with its probability p, as (a, p).

As seen from subsection 8.1.1, the DNS BAA SMG has three players: the attacker (AS), the defender (DFD), and the victim or the client server (CS). From Fig. 8.2, we see that AS and DFD have nondeterministic actions. In the state $AS : S_{init}$, the AS nondeterministically chooses the number of `zombies`, in the state $DFD : S_2$, the DFD nondeterministically decides to turn on or to turn off the countermeasures, and in the state $DFD : S_3^{cm\_on}$, the DFD nondeterministically chooses a countermeasure from FTR, RND, AGR, RDR, or AGF. However, the CS's actions are probabilistic, as it receives legitimate and bogus packets with probabilities that reflect the selected countermeasure, the

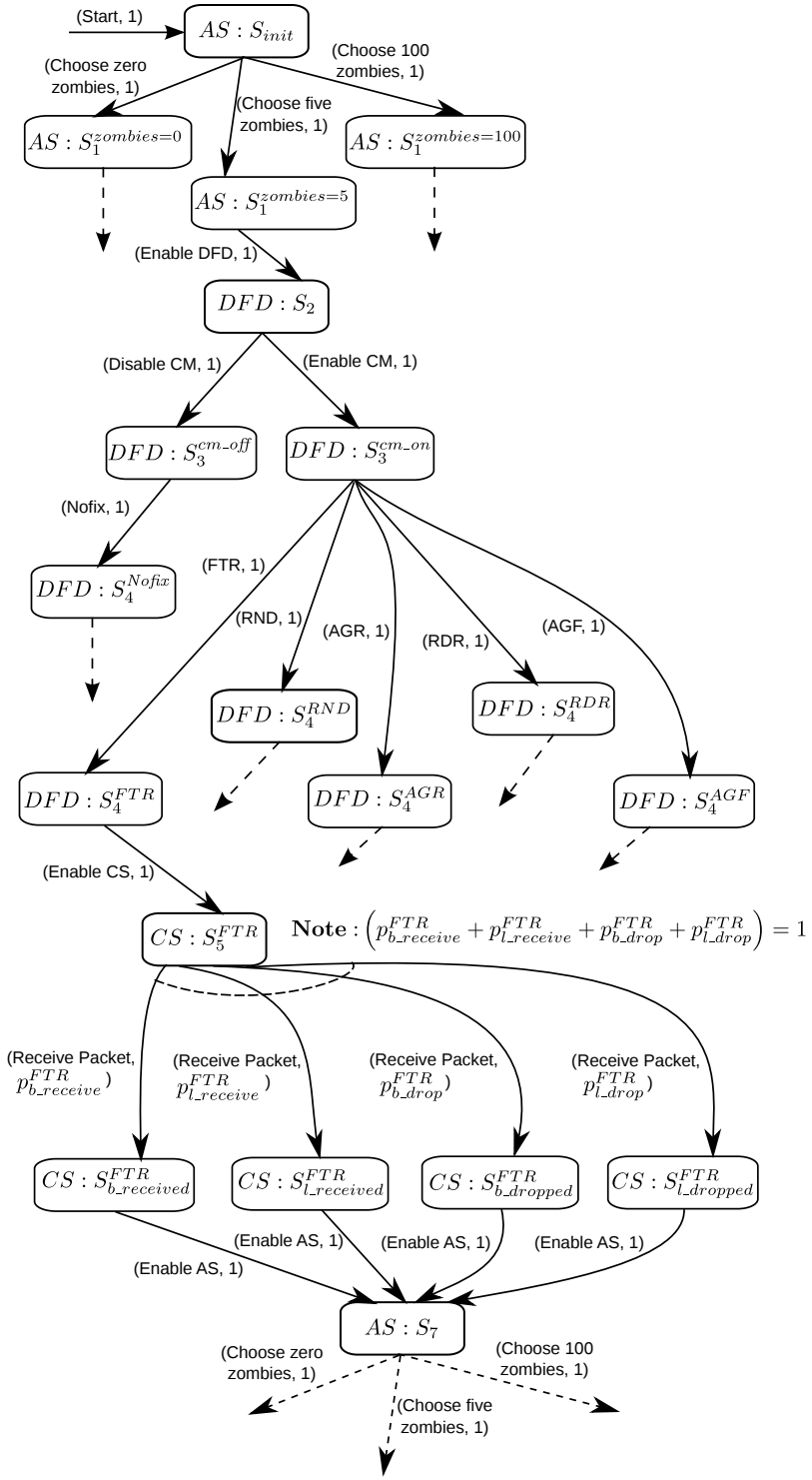


Figure 8.2: The DNS BAA SMG: Player P_i controlling a state S_j is represented as $P_i : S_j$. The label on each transition denotes the model action a, along with its probability p, as (a, p).

arrival rates of the legitimate and bogus DNS packets, and its bandwidth. We observe that in the state $CS : S_5^{FTR}$, the CS executes action *Receive Packet* and with probability $p_{b\_receive}^{cm}$, a bogus packet is received and the next state is $CS : S_{b\_received}^{FTR}$, with probability $p_{l\_receive}^{cm}$, a legitimate packet is received and the next state is $CS : S_{l\_received}^{FTR}$, with probability $p_{b\_drop}^{cm}$, a bogus packet is dropped and the next state is $CS : S_{b\_dropped}^{FTR}$, and with probability $p_{l\_drop}^{cm}$, a legitimate packet is dropped and the next state is $CS : S_{l\_dropped}^{FTR}$, such that $\left(p_{b\_receive}^{cm} + p_{l\_receive}^{cm} + p_{b\_drop}^{cm} + p_{l\_drop}^{cm}\right) = 1$. We now derive expressions for the (i) probability $p_{b\_receive}^{cm}$ with which the CS *receives* a *bogus* packet, (ii) probability $p_{l\_receive}^{cm}$ with which the CS *receives* a *legitimate* packet, (iii) probability $p_{b\_drop}^{cm}$ with which the CS *drops* a *bogus* packet, and (iv) probability $p_{l\_drop}^{cm}$ with which the CS *drops* a *legitimate* packet.

As observed from Fig. 8.1, the DFD sits at the periphery of the CS's network. All traffic first passes through the DFD before reaching the CS. So, there are two places where a packet can be dropped:

1. At the DFD, if DFD uses FTR, RND, RDR, or AGF
2. At the CS, if CS's bandwidth is full

First, we compute the rate $R_{b\_in}^{cm}$ at which bogus packets pass through the DFD, the rate $R_{l\_in}^{cm}$ at which legitimate packets pass through the DFD, the rate $R_{b\_out}^{cm}$ at which bogus packets are dropped at the DFD, and the rate $R_{l\_out}^{cm}$ at which legitimate packets are dropped at the DFD as follows:

$$R_{b\_in}^{cm} = \begin{cases} R_b & \text{if } cm = \textit{Nofix}, AGR \\ R_b \cdot (1 - \texttt{df}) & \text{if } cm = FTR, AGF \\ R_b \cdot (1 - \texttt{rdf}) & \text{if } cm = RND, RDR \end{cases} \tag{8.1}$$

$$R_{l\_in}^{cm} = \begin{cases} R_l & \text{if } cm = \textit{Nofix} \\ R_l \cdot (1 - \texttt{fpf}) & \text{if } cm = FTR \\ R_l \cdot (1 - \texttt{rdf}) & \text{if } cm = RND \\ R_l \cdot 2^{\texttt{retries}} & \text{if } cm = AGR \\ R_l \cdot 2^{\texttt{retries}} \cdot (1 - \texttt{rdf}) & \text{if } cm = RDR \\ R_l \cdot 2^{\texttt{retries}} \cdot (1 - \texttt{fpf}) & \text{if } cm = AGF \end{cases} \tag{8.2}$$

$$R_{b\_out}^{cm} = \begin{cases} 0 & \text{if } cm = \textit{Nofix}, AGR \\ R_b \cdot \texttt{df} & \text{if } cm = FTR, AGF \\ R_b \cdot \texttt{rdf} & \text{if } cm = RND, RDR \end{cases} \tag{8.3}$$

$$R_{l\_out}^{cm} = \begin{cases} 0 & \text{if } cm = \textit{Nofix}, AGR \\ R_l \cdot \texttt{fpf} & \text{if } cm = FTR \\ R_l \cdot \texttt{rdf} & \text{if } cm = RND \\ R_l \cdot 2^{\texttt{retries}} \cdot \texttt{rdf} & \text{if } cm = RDR \\ R_l \cdot 2^{\texttt{retries}} \cdot \texttt{fpf} & \text{if } cm = AGF \end{cases} \tag{8.4}$$

Next, we compute the probability $p^{cm}_{b\_in}$ with which a bogus packet passes through the DFD and reaches the CS, the probability $p^{cm}_{l\_in}$ with which a legitimate packet passes through the DFD and reaches the CS, the probability $p^{cm}_{b\_out}$ with which a bogus packet is dropped at the DFD, and the probability $p^{cm}_{l\_out}$ with which a legitimate packet is dropped at the DFD as follows:

$$p^{cm}_{b\_in} = \begin{cases} \frac{R^{cm}_{b\_in}}{R_b+R_l} & \text{if } cm = Nofix, FTR, RND \\ \frac{R^{cm}_{b\_in}}{R_b+R_l \cdot 2^{\texttt{retries}}} & \text{if } cm = AGR, RDR, AGF \end{cases} \tag{8.5}$$

$$p^{cm}_{l\_in} = \begin{cases} \frac{R^{cm}_{l\_in}}{R_b+R_l} & \text{if } cm = Nofix, FTR, RND \\ \frac{R^{cm}_{l\_in}}{R_b+R_l \cdot 2^{\texttt{retries}}} & \text{if } cm = AGR, RDR, AGF \end{cases} \tag{8.6}$$

$$p^{cm}_{b\_out} = \begin{cases} 0 & \text{if } cm = Nofix, AGR \\ \frac{R^{cm}_{b\_out}}{R_b+R_l} & \text{if } cm = FTR, RND \\ \frac{R^{cm}_{b\_out}}{R_b+R_l \cdot 2^{\texttt{retries}}} & \text{if } cm = RDR, AGF \end{cases} \tag{8.7}$$

$$p^{cm}_{l\_out} = \begin{cases} 0 & \text{if } cm = Nofix, AGR \\ \frac{R^{cm}_{l\_out}}{R_b+R_l} & \text{if } cm = FTR, RND \\ \frac{R^{cm}_{l\_out}}{R_b+R_l \cdot 2^{\texttt{retries}}} & \text{if } cm = RDR, AGF \end{cases} \tag{8.8}$$

Once a packet passes through the DFD, CS would receive it subject to CS's bandwidth availability. So, $p^{cm}_{b\_receive}$ can be computed as the probability that bogus packet arrived at CS *AND* CS received it. The $p^{cm}_{b\_drop}$ can be computed as (the probability that a bogus packet is dropped by DFD) *OR* (the probability that bogus packet arrives at CS *AND* is dropped by CS because of bandwidth congestion). Similarly, we can compute $p^{cm}_{l\_receive}$ and $p^{cm}_{l\_drop}$.

$$p^{cm}_{b\_receive} = \begin{cases} p^{cm}_{b\_in} \cdot \left(\frac{\texttt{BW}}{R^{cm}_{b\_in}+R^{cm}_{l\_in}} \right) & \text{if } \texttt{BW} < R^{cm}_{b\_in} + R^{cm}_{l\_in} \\ p^{cm}_{b\_in} & \text{otherwise} \end{cases} \tag{8.9}$$

$$p^{cm}_{l\_receive} = \begin{cases} p^{cm}_{l\_in} \cdot \left(\frac{\texttt{BW}}{R^{cm}_{b\_in}+R^{cm}_{l\_in}} \right) & \text{if } \texttt{BW} < R^{cm}_{b\_in} + R^{cm}_{l\_in} \\ p^{cm}_{l\_in} & \text{otherwise} \end{cases} \tag{8.10}$$

$$p^{cm}_{b\_drop} = \begin{cases} p^{cm}_{b\_out} + p^{cm}_{b\_in} \cdot \left(1 - \frac{\texttt{BW}}{R^{cm}_{b\_in}+R^{cm}_{l\_in}} \right) & \text{if } \texttt{BW} < R^{cm}_{b\_in} + R^{cm}_{l\_in} \\ p^{cm}_{b\_out} & \text{otherwise} \end{cases} \tag{8.11}$$

$$p^{cm}_{l\_drop} = \begin{cases} p^{cm}_{l\_out} + p^{cm}_{l\_in} \cdot \left(1 - \frac{\texttt{BW}}{R^{cm}_{b\_in}+R^{cm}_{l\_in}} \right) & \text{if } \texttt{BW} < R^{cm}_{b\_in} + R^{cm}_{l\_in} \\ p^{cm}_{l\_out} & \text{otherwise} \end{cases} \tag{8.12}$$

8.1.4 Attacker's and defender's payoffs and optimal attack and defense strategies

As described in subsection 8.1.1, to ensure that we have a two-player game, we divide the three players into two groups or coalitions: first group contains only the AS (attacker) and second group contains the DFD (defender) and the CS (victim). The DNS BAA game is played between these two coalitions, which are adversaries of each other.

Payoffs and net benefits: When we performed cost-benefit analysis of DNS cache-poisoning countermeasures (Chapter 5) and DNS BAA countermeasures (Chapter 6), we computed net benefits for each countermeasure. Then, we chose the countermeasure that offered the highest net benefit. The net benefit, therefore, indicates the desirability of a countermeasure. Similarly, for a given player, the payoff indicates the desirability of a game's outcome for that player. In a game, a player tries to generate a strategy that maximizes its payoff. So, we see that both net benefit and payoff serve a similar purpose. They both signify desirability of an activity, such as the choice of countermeasures. So, we would like to define the attacker and defender payoffs in a way similar to the net benefits defined for BAA countermeasures. However, certain limitations of PRISM rewards prevent us from defining such payoffs.

From section 6.2 and section 6.3, we see that we first computed individual benefits and costs for the entire duration of the experiment and then we subtracted the total cost from the total benefit to obtain the net benefit. However, PRISM rewards allow us to assign rewards to model's states or transitions. A typical PRISM reward definition thus looks like:

```
rewards "rewardDef"
 <condition> : <rewardValue>;
endrewards
```

where a numeric reward of `<rewardValue>` is assigned to a state that is identified by the condition `<condition>`. By writing a reward-based property

```
 <<P1>> Rmax{"rewardDef"}=? [F maxTime]
```

we can ask PRISM-games to generate a strategy for player `P1` such that it maximizes expected accumulated values of reward `rewardDef` until `time` reaches `maxTime`. Let us assume that we define a payoff using the net benefit definition from section 6.2. We could then define a reward structure as follows:

```
rewards "NetBenefit"
 <condition_1> : B1;
 <condition_2> : B2;
 <condition_3> : -C1;
endrewards
```

If we ask PRISM-games to identify a strategy that maximizes this payoff, then PRISM-games identifies a strategy that maximizes $nbf_1 + nbf_2 + \ldots + nbf_{maxTime}$, where nbf_t = net benefit computed at time t, i.e., the instantaneous value of the net benefit. Actually, the net

benefit should have been computed only at the end of the experiment. This is possible only if we use counters to explicitly maintain the count of the legitimate packets received over the duration of the experiment, the count of legitimate packets dropped over the duration of the experiment, and so on. Keeping such counters increases the model's state-space, which makes this approach infeasible. Therefore, we use simpler payoff definitions that consider only a single benefit or cost metric.

Attacker's payoffs: We define two *payoff* functions for the attacker, because we want to study how a payoff definitions affect the generated optimal strategies.

- $Payoff_1$: Maximize the difference between legitimate packets dropped per zombie and legitimate packets received per zombie.

```
rewards "AttackerPayoff_1"
 sched=0 & legitPacketDropped & zombies>0 : 1/zombies;
 sched=0 & legitPacketReceived & zombies>0 : -1/zombies;
 sched=0 & legitPacketDropped & zombies=0 : 1;
 sched=0 & legitPacketReceived & zombies=0 : -1;
endrewards
```

  If `zombies` > 0, then we assign a positive unit reward per zombie, when a legitimate packet is dropped and assign a negative unit reward per zombie, when a legitimate packet is received. When `zombies` = 0, i.e., when the attack is paused, we assign a positive unit reward if a legitimate packet is dropped and assign a negative unit reward if a legitimate packet is received.

- $Payoff_2$: Maximize the difference between bogus packets received and legitimate packets received

```
rewards "AttackerPayoff_2"
 sched = 0 & legitPacketReceived : -1;
 sched = 0 & bogusPacketReceived : 1;
endrewards
```

  We can see that the attacker $payoff_2$ is related to benefit B_1 defined for the BAA CTMC model (section 6.2). The attacker wants to maximize the difference between bogus packets received and legitimate packets received. So, the defender tries to minimize such difference. If the accumulated reward value is positive, then it means that the number of bogus packets received is twice the number of the legitimate packets received, i.e., benefit B_1(percentage of legitimate packets processed among total packets processed) is less than 50.

Defender's payoffs: Since our game is a zero-sum game, the defender's payoff is exactly opposite to the attacker's payoff. The defender always chooses a move that would be most disadvantageous to the attacker. So, we do not need to explicitly define the defender's payoffs.

Generating optimal attack and defense strategies: To generate the optimal attack and defense strategies, we define rPATL reward-based properties for the coalition of players containing the AS. These reward-based properties are of the form:

time	rdf=0.99, retries=6		rdf=0.99, retries=2		rdf=0.9, retries=6		rdf=0.9, retries=2	
	zombies	cm	zombies	cm	zombies	cm	zombies	cm
1	75	FTR	75	FTR	75	FTR	75	FTR
2	75	FTR	75	FTR	75	FTR	75	FTR
3	0	Nofix	0	AGR	0	Nofix	0	AGR
4	75	FTR	75	FTR	75	FTR	75	FTR
5	75	FTR	75	FTR	75	FTR	75	FTR
6	75	FTR	75	FTR	75	FTR	75	FTR
7	75	FTR	75	FTR	75	FTR	75	FTR
8	0	Nofix	0	AGR	0	Nofix	0	AGR
9	75	FTR	75	FTR	75	FTR	75	FTR
10	75	FTR	75	FTR	75	FTR	75	FTR
11	75	FTR	75	FTR	75	FTR	75	FTR
12	75	FTR	75	FTR	75	FTR	75	FTR
13	0	Nofix	0	AGR	0	Nofix	0	AGR
14	75	FTR	75	FTR	75	FTR	75	FTR
15	75	FTR	75	FTR	75	FTR	75	FTR
16	75	FTR	75	FTR	75	FTR	75	FTR
17	75	FTR	75	FTR	75	FTR	75	FTR
18	0	Nofix	0	AGR	0	Nofix	0	AGR
19	75	FTR	75	FTR	75	FTR	75	FTR
20	75	FTR	75	FTR	75	FTR	75	FTR

Table 8.1: Result set 1 — Optimal attack and defense strategies generated for attacker $Payoff_1$

```
<<AS>> R{"AttackerPayoff_n"}max=? [ F time=maxTime ]
```

Upon evaluation of such properties, PRISM-games returns the expected accumulated reward value and the optimal strategies for the two coalitions of players, viz. the coalition containing the AS and the coalition containing the DFD and the CS.

8.2 Experimental results

Using different values of `rdf` and `retries`, we generate optimal attack and defense strategies for the both attacker payoffs, *Payoff*$_1$ and *Payoff*$_2$. We set R_l to 100 packets per second, `AF` to 15.31, R_b to `zombies` $\cdot$ 10 $\cdot$ `AF` packets per second, `BW` to 458 packets per second, `MAX_SUCC_ATTACKS` to 5 seconds, `ATTACKER_LATENCY` to 2 seconds, `df` to 0.9, and `fpf` to 0.1. We set `rdf` to 0.99 and 0.9, so that we cover the cases where `rdf` $>$ `df` and `rdf` $\leq$ `df`. We set `retries` to 6 (high) and 2 (low). Finally, we set the `maxTime` to the maximum value that is permitted by the constraints imposed by model's state-space size. In order to prevent the state-space explosion, we set `maxTime` to 20 seconds.

Result Set 1 — Optimal attack and defense strategies for $Payoff_1$: Table 8.1 presents the optimal attack and defense strategies generated for the attacker $Payoff_1$. From Table 8.1, we observe that the attacker never attacks for five continuous seconds, so as to escape the latency. We observe that though the attacker can use up to 100 zombies, it uses

time	rdf=0.99, retries=6		rdf=0.99, retries=2		rdf=0.9, retries=6		rdf=0.9, retries=2	
	zombies	cm	zombies	cm	zombies	cm	zombies	cm
1	100	RDR	100	RDR	100	AGR	25	FTR
2	100	RDR	100	RDR	100	AGR	25	FTR
3	100	RDR	100	AGF	0	Nofix	0	AGR
4	100	RDR	0	AGR	100	AGR	25	FTR
5	0	Nofix	100	RDR	100	AGR	25	FTR
6	100	RDR	100	RDR	100	AGR	25	FTR
7	100	RDR	100	RDR	100	AGR	25	FTR
8	100	RDR	0	AGR	0	Nofix	0	AGR
9	100	RDR	100	RDR	100	AGR	25	FTR
10	0	Nofix	100	RDR	100	AGR	25	FTR
11	100	RDR	100	RDR	100	AGR	25	FTR
12	100	RDR	100	RDR	100	AGR	25	FTR
13	100	RDR	0	AGR	0	Nofix	0	AGR
14	100	RDR	100	RDR	100	AGR	25	FTR
15	0	Nofix	100	RDR	100	AGR	25	FTR
16	100	RDR	100	RDR	100	AGR	25	FTR
17	100	RDR	100	RDR	100	AGR	25	FTR
18	100	RDR	0	AGR	0	Nofix	0	AGR
19	0	Nofix	100	RDR	100	AGR	25	FTR
20	100	RDR	100	RDR	100	AGR	25	FTR

Table 8.2: Result set 2 — Optimal attack and defense strategies generated for attacker $Payoff_2$

at most 75 zombies. To understand why this happens, we need to revisit the definition of the attacker $Payoff_1$. We see that the attacker $Payoff_1$ assigns unit reward *per zombie* when a legitimate packet is dropped and a unit negative reward *per zombie* when a legitimate packet is received. This causes the attacker to use just enough number of zombies that are required to maximize its payoff, since using more zombies than required may, in fact, reduce the payoff. So, the attacker uses 75 zombies as opposed to 100 zombies. We observe that when the attacker uses high number of `zombies`, e.g., 75, then the defender uses FTR, as FTR drops high percentage of bogus packets with minimum false-positives. But, when the attacker pauses the attack, then the defender either turns off the countermeasures or uses AGR, because both Nofix and AGR ensure that as many legitimate packets are received without any packets getting dropped as false-positives. We see that when high number of `retries`, e.g., `retries` = 6, are used, then the defender prefers Nofix over AGR, but when the number of retries are low, then the defender chooses AGR over Nofix. This happens because, when `retries` = 2, the total rate at which the legitimate packets flow = $R_l \cdot 2^2 = 400$ packets per second. This rate is higher than the base value of R_l (100 packets per second), but it is still lower than the victim's bandwidth (`BW`) of 458 packets per second. So, the victim is able to accept all 400 packets sent per second, which is certainly better than accepting merely 100 packets per second, as accepting 400 packets per second rapidly increases the proportion of legitimate traffic received among total legitimate packets sent. In fact, AGR is preferred over Nofix as long as $R_l \cdot 2^{\texttt{retries}} \leq$ `BW`. However, when `retries` = 6, the total rate at which the legitimate packets flow = $R_l \cdot 2^6 = 6400$ packets per second, which is higher than the victim's bandwidth. So, if in this case, the victim employs AGR, then more legitimate packets are going to be dropped because of the bandwidth congestion. This can be avoided by using Nofix, as the bandwidth is sufficient to accept all 100 packets sent per second.

We also observe that none of the optimal strategies generated used RND or RDR. This happens, because when RND is used, the values of probability $p^{RND}_{l\_receive}$ that a legitimate packet is received are always less than the corresponding probability values observed for FTR and AGR. Moreover, $p^{RND}_{l\_receive} = p^{Nofix}_{l\_receive}$, because RND drops `rdf` of legitimate and bogus traffic. So, selecting RND or RDR does not help the victim to increase the fraction of legitimate packets received. The victim, therefore, chooses either FTR or Nofix.

Result Set 2 — Optimal attack and defense strategies for $Payoff_2$: Table 8.2 presents the optimal attack and defense strategies generated for the attacker $Payoff_2$. From Table 8.2, we again see that the attacker never attacks for five continuous seconds. We also see that the defender does use RDR. RND's primary responsibility is to rate limit the incoming traffic, and while doing so drop as many bogus packets as possible. When we use the attacker *Payoff*$_2$, then the attacker tries to ensure that more bogus packets are received than the legitimate packets, so it prefers to use as many zombies as possible, which is 100. The defender, on the other hand, tries to drop as many bogus packets as possible. So, when `rdf` $>$ `df`, then the defender uses RDR (RND + AGR), since RDR reduces the number of bogus packets received with a greater probability than FTR. If `rdf` $\leq$ `df`, then victim uses AGR or FTR, because now AGR and FTR drop many bogus packets and allow a high number of legitimate packets to be received. We also see that when the attacker pauses the attack, then the defender uses FTR if the number of `retries` are high, else it uses AGR.

Chapter 9

Related Work

Several works present formal analysis of Denial of Service (DoS), however, none of them analyze the DNS BAA. Also, to the best of our knowledge, we are the first to formally model DNS cache poisoning and perform cost-benefit analysis of its countermeasures.

A noteworthy alternative to our work is [53], which uses a game-theoretic framework to study bandwidth attacks. The attack is modeled as a *traffic injection game* between the attacker and the defender. Defender's effectiveness is analyzed from the payoffs of different strategies used in the traffic injection game, however, only one type of defense is considered, viz., the filtering.

A DoS-resistant 3-way handshaking in the Transmission Control Protocol is modeled in [3] using the probabilistic rewriting logic. Instead of using a formal stochastic model like a CTMC, a timed probabilistic model is generated from the developed algebraic specification, which is then analyzed by statistical model checking. This simulation-based analysis cannot be as accurate as the probabilistic model checking approach. Statistical model checking is also used in [6] for analyzing the ASV protocol as a DoS countermeasure.

A Discrete Time Markov Chain model is presented in [10], quantifying DoS threats against an authentication protocol. In [43], the authors present a cost-based analysis to compare the cost imposed on the attacker against the cost for defending honest participants in a protocol under DoS attack. This approach is instantiated into a probabilistic model checking framework. CTMC based analysis with reward properties is proposed in [26] to analyze a DDoS attack against the Mobile IP and Seamless IP diversity based Generalized Mobility Architecture. Both [10, 26] provide no analysis and comparison of related countermeasures.

An amplification attack is modeled in [49] using states where some measure comparisons hold true. This measure checking is implemented with rewriting logic, an executable specification that is model checked in the Maude tool. This approach tries to automatically look up for known attacks and verify that a patch for an attack achieves its aim, but no comparison between alternative solutions is supported.

In [28], the authors propose the Highly-Available Redundantly-Distributed DNS or the HARD-DNS, a distributed network of DNS resolvers to provide a robust protection against DNS cache poisoning and DDoS. HARD-DNS uses quorum techniques to provide reliable answers in presence of cache poisoning and IP-cloaking to protect the connection

between resolvers and HARD-DNS. Authors evaluate performance of HARD-DNS by deploying it on the PlanetLab [11], an open-source platform for testing large-scale services. Such simulation-based performance valuation usually costs considerably more than our formal model-based approach. Also, formal model-based results are more reliable than any simulation-based results.

Game-theoretic approaches have been used in past to analyze the DoS and the DDoS attacks. These works model DoS or the DDoS as a two-player games played between an attacker and a defender. The defenders use filtering and rate limiting as countermeasures against the attack. Majority of these works [53, 50, 67, 12, 2] analyze effectiveness of individual countermeasures, however, [66] studies multi-layer protection where filtering, rate-limiting, and bandwidth capacity extension are used in conjunction to combat DDoS. The payoff functions are defined for attacker and defender, which reflect the benefits obtained and the costs incurred. The typical metrics used as benefits and costs are: the bandwidth share utilized by the legitimate traffic, the number of legitimate packets incorrectly dropped, the number of bogus packets allowed to pass through the filter, and the costs associated with adding more bandwidth or using more number of zombies. The payoffs are defined as functions of: various thresholds and drop rates (DR) used by filtering mechanism, rate at which legitimate packets arrive (LR), number of zombies used by the attacker (Z), rate at which each zombie sends bogus traffic (BR), and the manner in which the attack traffic is generated (ATG), i.e., whether the attack traffic is sent continuously in bursts. Once payoff functions are defined, they are used to determine the Nash equilibria strategies for attacker and defender. Nash equilibria are computed using various game solvers and the results are verified by simulating the attack using network simulation tools such as NS2. The Nash equilibria return the values of DR that would maximize defender's payoff and values of values of Z, BR, and ATG that would maximize attacker's payoff. Neither defender nor attacker has any incentive to deviate from the Nash equilibria.

Some similarities do exist between our work and the earlier works, especially in the choice of the model parameters and the types of countermeasures. However, our work is different in several aspects. Along with FTR and RND, we analyze AGR too. We also analyze RDR and AGF, the composite countermeasures formed by combining FTR, RND, and AGR. Our two-player model of the DNS BAA allows generation of more interesting optimal attack and defense strategies where the number of `zombies` chosen to launch the attack and the countermeasures chosen to prevent the attack are varied over the duration of the experiment, instead of using a fixed number of `zombies` and a fixed countermeasure. We design the two-player game for the DNS BAA using PRISM-games, which has the ability to generate optimal strategies. Therefore, we do not need to use a separate game solver to find the optimal strategies. Also, reward-based properties in PRISM offer a convenient way to define payoff functions. So, we can easily define multiple payoff functions and generate optimal strategies for them.

Chapter 10

Conclusions

We used the probabilistic model checker PRISM to formally model and analyze the DNS cache poisoning and the bandwidth amplification attacks, both of which pose a major threat to the reliability and availability of DNS. We also modeled various countermeasures designed to prevent these attacks from occurring, and used the resulting models to perform a countermeasure cost-benefit analysis.

Our cost-benefit analysis of the two cache poisoning countermeasures revealed that the superior protection offered by RDQ over PRAND comes at the cost of increased bandwidth usage. RDQ and PRAND are short-term fixes for the DNS cache poisoning problem, while the cryptography-based DNSSec offers a long-term solution. We therefore investigated if DNSSec is vulnerable against other DNS attacks, notably BAA. We formally proved that DNSSec is, in fact, more vulnerable than DNS to a BAA.

Our cost-benefit analysis of the BAA countermeasures showed a significant variation in the performance of five BAA countermeasures, with AGF offering the highest net benefit. Moreover, model-checking times did not exceed four seconds for all reward and probabilistic reachability properties considered, for both cache poisoning and BAA. In the process, we developed a general cost-benefit analysis framework for probabilistic systems based on probabilistic model checking. This framework helped us to think about new and efficient ways to counter the threat posed to DNS by BAA, as seen in the development of two superior BAA countermeasures; viz., RDR and AGF, each of which is formed by combining two basic countermeasures.

We successfully modeled the DNS BAA as a two-player, turn-based, zero-sum stochastic game using PRISM-games, where the attacker tries to flood the victim's bandwidth with large-sized unwanted responses and the defender tries to mitigate the attack by applying a countermeasure. Our stochastic model of the DNS BAA allowed us to generate the optimal attack and defense strategies where the attacker tries to either maximize the difference between legitimate packets dropped per zombie and legitimate packets received per zombie or maximize the difference between bogus packets received and legitimate packets received, while the defender chooses the best possible countermeasures to oppose the attacker. Our results showed that the generated optimal attack and defense strategies depend on the countermeasure parameters, the payoff function used, and any scheduled interruptions that the attacker may have to experience.

We believe that our stochastic game-based model of the DNS BAA can be further

enhanced by allowing the PRISM-games engine to choose the optimal values of countermeasure parameters as well. As it can be seen from section 8.2, we generated the optimal attack and defense strategies by setting `rdf` to 0.99 and 0.9 and by setting `retries` to 6 and 2. We could extend our model, so that PRISM-games itself chooses the `rdf` and `retries` values from a given range of values. Such extension would be interesting, as it would allow the PRISM-games to identify the optimal countermeasure parameter values, thereby generating more effective defense strategies.

We could also extend the stochastic game-based model of the DNS BAA to generate the optimal attack and defense strategies for more complex payoff functions. For example, the attacker may try to maximize the percentage of legitimate packets dropped per zombie. This payoff function needs to be evaluated over the entire model run. So, it is challenging to define such payoff functions using PRISM-games rewards, because PRISM-games allows us to assign rewards only to individual states of the model. A possible way to extend our model to support this payoff function is by maintaining *counters* in the model to keep track of the number of legitimate packets received and the number of legitimate packets dropped. The reward could then be defined as

```
rewards "AttackerPayoff"
 time=maxTime: ((numLegitPacketsReceived)/
  (numLegitPacketsReceived + numLegitPacketsDropped))
  /zombies
endrewards
```

However, usage of counters increases the state-space a lot, and therefore, it may be infeasible. Also, if the number of zombies change over the course of the model run, then only the latest value of number of zombies would be used during the reward evaluation, which would invalidate the reward definition. This can be avoided by using the average number of zombies utilized over the course of the model run, which would require us to use another counter variable, thereby increasing the model's already big state-space.

Bibliography

[1] Monitoring DNS server performance. Technical report, Microsoft `http://technet.microsoft.com/en-us/library/cc778608(WS.10).aspx`.

[2] A. Agah and S. Das. Preventing DoS attacks in wireless sensor networks: A repeated game theory approach. *International Journal of Network Security*, pages 145–153, 2007.

[3] G. Agha, M. Greenwald, C. A. Gunter, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of DoS using probabilistic rewrite theories. In *Proc. IEEE Int. Workshop on Foundations of Computer Security (FCS'05)*, 2005.

[4] R. Aitchison. *Pro DNS and BIND 10*. Apress, Berkely, CA, USA, 1st edition, 2011.

[5] N. Alexiou, T. Deshpande, S. Basagiannis, P. Katsaros, and S. A. Smolka. Formal analysis of the Kaminsky DNS cache-poisoning attack using probabilistic model checking. In *IEEE 12th Int. Symposium on High-Assurance Systems Engineering (HASE'10)*, pages 94–103, 2010.

[6] M. AlTurki, J. Meseguer, and C. A. Gunter. Probabilistic modeling and analysis of DoS protection for the ASV protocol. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 234:3–18, Mar 2009.

[7] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol modifications for the DNS security extensions. RFC 4035, March 2005.

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[9] E. Bartocci, R. Grosu, P. Katsaros, C. R. Ramakrishnan, and S. A. Smolka. Model repair for probabilistic systems. In *Proc. 17th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'11)*, volume 6605 of *LNCS*, pages 326–340, 2011.

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

[11] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. 1st Conference on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 19–19, 2004.

[12] H.S. Bedi, S. Roy, and S. Shiva. Game theory-based defense mechanisms against DDoS attacks on TCP/TCP-friendly flows. In *Proc. IEEE Symposium on Computational Intelligence in Cyber Security (CICS'11)*, pages 129–136, 2011.

[13] D. Bernstein. DNSCurve: Usable security for DNS. Jun 2009 (`http://dnscurve.org/`).

[14] A. Boardman, D. Greenberg, A. Vining, and D. Weimer. *Cost Benefit Analysis: Concepts and Practice*. Prentice Hall, 3rd edition, 2006.

[15] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. Automatic verification of competitive stochastic systems. *Formal Methods in System Design*, 2013. To appear.

[16] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. PRISM-games: A model checker for stochastic multi-player games. In N. Piterman and S. Smolka, editors, *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, volume 7795 of *LNCS*, pages 185–191. Springer, 2013.

[17] T. Chen, M. Kwiatkowska, D. Parker, and A. Simaitis. Verifying team formation protocols with probabilistic model checking. In *Proc. 12th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA XII 2011)*, volume 6814 of *LNCS*, pages 190–297. Springer, 2011.

[18] A. Cowperthwaite and A. Somayaji. The futility of DNSSec. In *Proc. 5th Annual Symp. Information Assurance (ASIA'10)*, pages 2–8, 2010.

[19] D. Dagon, M. Antonakakis, P. Vixie, T. Jinmei, and W. Lee. Increased DNS forgery resistance through 0x20-bit encoding: SecURItY viA LeET QueRieS. In *Proc. the 15th ACM conference on Computer and communications security*, CCS '08, pages 211–222, 2008.

[20] T. Deshpande, P. Katsaros, S. Basagiannis, and S. A. Smolka. Formal analysis of the DNS bandwidth amplification attack and its countermeasures using probabilistic model checking. In *Proc. 13th IEEE International Symposium on High-Assurance Systems Engineering (HASE'11)*, pages 360–367, 2011.

[21] DNS forgery (`http://dnscurve.org/forgery.html`).

[22] The Measurement Factory. Dns survey: Open resolvers (`http://dns.measurement-factory.com/surveys/openresolvers.html`).

[23] V. Forejt, M. Kwiatkowska, G. Norman, and D. Parker. Automated verification techniques for probabilistic systems. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems (SFM'11)*, volume 6659 of *LNCS*, pages 53–113. Springer, 2011.

[24] S. Ben Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical bandwidth sharing: a study of congestion at flow level. *ACM SIGCOMM Computer Communication Review*, 31(4):111–122, 2001.

[25] S. Friedl. An illustrated guide to the Kaminsky DNS vulnerability, Aug 2008 (`http://unixwiz.net/techtips/iguide-kaminsky-dns-vuln.html`).

[26] S. Fu and M. Atiquzzaman. Survivability evaluation of SIGMA and mobile IP. *Wireless Personal Communications: An International Journal*, 43:933–944, Nov 2007.

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

[28] C. Gutierrez, R. Krishnan, R. Sundaram, and Fangfei Zhou. HARD-DNS: Highly-available redundantly-distributed DNS. In *Proc. Military Communications Conference, 2010 (MILCOM'10)*, pages 1343 –1348, Nov 2010.

[29] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition, 2011.

[30] J. Hoy. Anti DNS spoofing - extended query id (XQID). Apr 2008 (`http://www.jhsoft.com/dns-xqid.htm`).

[31] S. Ioannidis, K. G. Anagnostakis, J. Ioannidis, and A. D. Keromytis. xPF: packet filtering for low-cost network monitoring. In *Workshop on High Performance Switching and Routing Merging Optical and IP Technologies, 2002*, pages 116–120, 2002.

[32] V. Jacobson. Minutes of the performance working group. In *Proc. Cocoa Beach Internet Engineering Task Force*, Apr 1989.

[33] C. Jin, H. Wang, and K. G. Shin. Hop-count filtering: an effective defense against spoofed DDoS traffic. In *Proc. 10th ACM Conf. on Computer and communications security (CCS'03)*, pages 30–41, 2003.

[34] O. Johansson-Stenman. Distributional weights in cost-benefit analysis: Should we forget about them? *Land Economics*, 81(3):337–352, 2005.

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

[36] G. Kambourakis, T. Moschos, D. Geneiatakis, and S. Gritzalis. Detecting DNS amplification attacks. In *Critical Information Infrastructures Security*, volume 5141 of *LNCS*, pages 185–196. 2008.

[37] S. Khanna, S. S. Venkatesh, O. Fatemieh, F. Khan, and C. A. Gunter. Adaptive selective verification. In *Proc. 27th IEEE Conf. on Computer Communications (INFOCOM'08)*, pages 529–537, Apr 2008.

[38] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270, 2007.

[39] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

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

[41] A. Mankin. Random drop congestion control. In *Proc. ACM Symposium on Communications Architectures & Protocols*, SIGCOMM'90, pages 1–7, 1990.

[42] A. McIver and C. Morgan. Results on the quantitative mu-calculus qMu. *ACM Transactions on Computational Logic*, 8(1), 2007.

[43] C. Meadows. A formal framework and evaluation method for network denial of service. In *Proc. 12th IEEE Workshop on Computer Security Foundations (CSFW'99)*, 1999.

[44] A. Neyman and S. Sorin. *Stochastic Games and Applications*. Nato Science Series. 2003.

[45] T. Peng, C. Leckie, and K. Ramamohanarao. Protection from distributed denial of service attacks using history-based IP filtering. In *Proc. IEEE Int. Conf. on Communications, 2003 (ICC'03)*, volume 1, pages 482–486, May 2003.

[46] R. Perdisci, M. Antonakakis, X. Luo, and W. Lee. WSEC DNS: Protecting recursive dns resolvers from poisoning attacks. In *Proc. IEEE/IFIP International Conference on Dependable Systems Networks (DSN '09)*, pages 3–12, Jul 2009.

[47] M. Prince. How to launch a 65Gbps DDoS, and how to stop one. Oct 2012 (`http://blog.cloudflare.com/65gbps-ddos-no-problem`).

[48] M. Prince. Deep inside a DNS amplification DDoS attack. Oct 2012 (`http://blog.cloudflare.com/deep-inside-a-dns-amplification-ddos-attack`).

[49] R. Shankesi, M. AlTurki, R. Sasse, C. Gunter, and J. Meseguer. Model-checking DoS amplification for VoIP session initiation. In *Proc. 14th European Symposium on Research in Computer Security*, pages 390–405, 2009.

[50] P. Shi and Y. Lian. Game-theoretical effectiveness evaluation of DDoS defense. In *Proc. the Seventh International Conference on Networking*, ICN '08, pages 427–433, 2008.

[51] G. Shively and M. Galopin. An overview of benefit-cost analysis (http://www.agecon.purdue.edu/staff/shively/COURSES/AGEC406/reviews/bca.htm).

[52] M. Shor. Payoff. In *Dictionary of Game Theory Terms, Game Theory.net*.

[53] M. E. Snyder, R. Sundaram, and M. Thakur. A game-theoretic framework for bandwidth attacks and statistical defenses. In *Proc. the 32nd IEEE Conference on Local Computer Networks*, LCN '07, pages 556–566. IEEE Computer Society, 2007.

[54] E. Somanathan. Valuing lives equally: Distributional weights for welfare analysis. *Economics Letters*, 90(1):122–125, 2006.

[55] H. Sun, Y. Zhaung, and H. J. Chao. A principal components analysis-based robust DDoS defense system. In *Proc. IEEE Int. Conf. on Communications, 2008 (ICC'08)*, pages 1663–1669, May 2008.

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

[57] J. Trostle, B. Van Besien, and A. Pujari. Protecting against DNS cache poisoning attacks. In *Proc. 6th IEEE Workshop on Secure Network Protocols (NPSec'10)*, pages 25–30, Oct 2010.

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

[59] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. *ACM Transactions on Computer Systems*, 28:3:1–3:54, Aug 2010.

[60] T. Watkins. An introduction to cost benefit analysis. (http://www.sjsu.edu/faculty/watkins/cba.htm).

[61] Wikipedia. Pearson product-moment correlation coefficient — wikipedia, the free encyclopedia (http://en.wikipedia.org/w/index.php?title=Pearson_product-moment_correlation_coefficient&oldid=489640269), 2012. [Online; accessed 6-May-2012].

[62] Wikipedia. Nash equilibrium — wikipedia, the free encyclopedia (http://en.wikipedia.org/w/index.php?title=Nash_equilibrium&oldid=560155666). 2013. [Online; accessed 18-June-2013].

[63] Wikipedia. Zero-sum game — wikipedia, the free encyclopedia (http://en.wikipedia.org/w/index.php?title=Zero%E2%80%93sum_game&oldid=557041044). 2013. [Online; accessed 18-June-2013].

[64] Z. Wu, M. Xie, and H. Wang. Swift: a fast dynamic packet filter. In *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*, pages 279–292, 2008.

[65] Y. Xuebiao, W. Xin, L. Xiaodong, and Y. Baoping. DNS measurements at the .CN TLD servers. In *Proc. 6th Int. Conf. on Fuzzy Systems and Knowledge Discovery (FSKD'09)*, volume 7, pages 540–545, Aug 2009.

[66] G. Yan, R. Lee, A. Kent, and D. Wolpert. Towards a bayesian network game framework for evaluating DDoS attacks and defense. In *Proc. the 2012 ACM conference on Computer and communications security (CCS'12)*, pages 553–566, 2012.

[67] W. Zang, P. Liu, and M. Yu. How resilient is the internet against DDoS attacks? a game theoretic analysis of signature-based rate limiting. In *International Journal of Intelligent Control and Systems*, volume 12, December.

Appendices

Appendix A

PRISM Code for CTMC Model of DNS Cache Poisoning

Model source code:

```
ctmc

// Popularity relates to the likelihood of IP address of a
// requested url to be cached in CS's cache.
const double popularity;

// Range of 16-bit query ids
const query_id=65536;

// Number of port id bits
const port_id_bits;

// Range of port ids as determined by port_id_bits
const port_id = pow(2, port_id_bits);

// Rate at which the AS sends bogus responses to the CS
const guess;

// Rate at which requests from DNS servers other than CS
// arrive at the DS
const other_legitimate_requests_rate;

// Number of non-authoritative servers (NASs) involved in
// the url resolution.
const NAS_count;

// CM is a normal desktop machine that requests a url to be
// resolved
```

```
module CM

request_sent : bool init false;
answer_received : bool init false;

[client_issues_url_resolution_request] request_sent = false
-> (request_sent'= true);

[forward_authoritative_answer_to_CM] answer_received =
   false
-> (answer_received'= true);

endmodule


// CS is the victim of the attack
module CS

// Time-to-live that determines validity of a cache entry.
// With probability of popularity/10, the cache entry is
// valid (ttl = 1) and with probability (1-popularity/10),
// the cache-entry has expired.
ttl : [0..2] init 2;

query_authoritative_server : bool init false;
query_non_authoritative_servers : bool init false;
queue : bool init false;
non_authoritative_servers_queue : bool init false;
authoritative_server_queue : bool init false;
queries_answered : bool init false;
correct_guess : bool init false;
valid_answer_received : bool init false;

// Cache is poisoned if AS correctly guesses <query id,
// port id> and the bogus answer arrives at CS before
// legitimate answer
corrupted_answer_received : bool init false;

// The authentic answer is received
answer_from_domain_received : bool init false;

// Client query is satisfied by a cached answer
[client_issues_url_resolution_request] queue = false &
   queries_answered = false &
   non_authoritative_servers_queue = false
```

```
-> popularity/10 : (queue'= true) & (ttl'=1) & (
   queries_answered'= true) & (valid_answer_received'= true
   );

// No cached answer is found.  So, clients's query would be
// resolved recursively.
[client_issues_url_resolution_request] queue = false &
   queries_answered = false &
   non_authoritative_servers_queue = false
-> (1 - popularity/10) : (queue'= true) & (ttl'=0) & (
   non_authoritative_servers_queue'= true);

[send_request_to_NAS] queue = true &
   non_authoritative_servers_queue = true &
   query_non_authoritative_servers = false
->(query_non_authoritative_servers'=true);

[receive_answer_from_NAS] query_non_authoritative_servers =
    true & authoritative_server_queue = false &
   correct_guess = false
-> (non_authoritative_servers_queue'= false) & (
   authoritative_server_queue'= true) & (
   query_non_authoritative_servers'=false);

[receive_answer_from_attacker_while_in_race_with_NAS]
   query_non_authoritative_servers = true &
   authoritative_server_queue = false & correct_guess =
   true
-> (non_authoritative_servers_queue'= false) & (
   query_non_authoritative_servers'=false) & (
   queries_answered'= true) & (answer_from_domain_received
   '=true) & (corrupted_answer_received'= true);

[send_request_to_DS] authoritative_server_queue = true &
   query_authoritative_server = false -> (
   query_authoritative_server'=true);

[receive_answer_from_DS_while_in_race_with_AS]
   query_authoritative_server = true & queries_answered =
   false & correct_guess=false
-> (authoritative_server_queue'= false) & (queries_answered
   '= true) & (query_authoritative_server'=false) & (
   answer_from_domain_received'=true) & (
   valid_answer_received'= true);
```

```
[receive_answer_from_AS_while_in_race_with_DS]
   authoritative_server_queue = true & queries_answered =
   false & correct_guess=true
-> (authoritative_server_queue'= false) & (queries_answered
   '= true) & (query_authoritative_server'=false) & (
   answer_from_domain_received'=true) & (
   corrupted_answer_received' = true);

[forward_authoritative_answer_to_CM] queries_answered =
   true & queue = true
-> (queue'= false) & (queries_answered'= false);

[Correct_Guess] correct_guess=false &
   query_authoritative_server=true
-> 1/(query_id*port_id) : (correct_guess'=true);

[Correct_Guess_In_Race_With_NAS] correct_guess=false &
   query_non_authoritative_servers=true
-> 1/(query_id*port_id) : (correct_guess'=true);

endmodule

// NAS represents all intermediate non-authoritative DNS
// servers including root, and com that are involved in the
// url resolution process
module NAS

[send_request_to_NAS] true -> true;

[receive_answer_from_NAS] true -> (1/(NAS_count-1)):true;

endmodule

// DS is the authoritative name server for the target
// domain google.com
module DS

authoritative_server_enabled: bool init false;

[send_request_to_DS] authoritative_server_enabled = false
  -> (authoritative_server_enabled'= true);

[receive_answer_from_AS_while_in_race_with_DS]
   authoritative_server_enabled = true -> (
   authoritative_server_enabled'= false);
```

```
[receive_answer_from_DS_while_in_race_with_AS]
   authoritative_server_enabled = true
-> 1/(other_legitimate_requests_rate) : (
   authoritative_server_enabled'= false);

endmodule

// AS is the authoritative DNS server for attacker's domain
// badguy.com.
module AS

attacker_enabled: bool init false;

[send_request_to_NAS] attacker_enabled = false -> (
   attacker_enabled'= true);

[receive_answer_from_NAS] attacker_enabled = true -> (
   attacker_enabled'= false);

[receive_answer_from_attacker_while_in_race_with_NAS]
   attacker_enabled = true -> (attacker_enabled'= false);

[send_request_to_DS] attacker_enabled = false -> (
   attacker_enabled'= true);

[receive_answer_from_DS_while_in_race_with_AS]
   attacker_enabled = true -> (attacker_enabled'= false);

[receive_answer_from_AS_while_in_race_with_DS]
   attacker_enabled = true -> (attacker_enabled'= false);

[Correct_Guess] attacker_enabled = true & guess > 0 ->
   guess: true;

[Correct_Guess_In_Race_With_NAS] attacker_enabled = true &
   guess > 0 -> guess: true;

endmodule
```

Properties file:

```
// Compute the attack probability that CS's cache is
// poisoned, i.e., CS receives a corrupted answer from AS
// before the authentic answer from DS arrives
P=? [ F corrupted_answer_received ]
```

Groovy script to determine countermeasure parameters for RDQ:

```
// The data file contains 30 values of attack
// probabilities p, with one entry per line.
data = new File('data.txt')

data.eachLine {line ->
        // Read each value of attack probability ''p''
        p = line.toDouble()

        // The probability ''q'' of successfully getting
           the
        // correct target IP address
        q = 1 - p

        // The upper bound for the number of retries is set
        // to 10000
        n_upper_bound = 10000

        // We initialize n_max to 1
        n_max = 1
        while(n_max < n_upper_bound){

                // p_noresponse is the probability of RDQ
                // needing another retry after the
                // n th retry (equation 6)
                p_noresponse = p * q * (p**(n_max-1) + q**(
                   n_max-1))

                // If p_noresponse is very close to 0, then
                // it means that we have found n_max
                if (p_noresponse * 1000 < 1){
                        // Now, we compute n_expected using
                        // the value of n_max that we have
                        // found
                        n_expected = 0
                        1.upto(n_max) { j ->
                                // We use equation 7
                                p_noresponse = p * q * (p
                                   **(j-1) + q**(j-1))
                                n_expected += j *
                                   p_noresponse
                        }
                        // Print the integer value of
                        // n_expected
```

```
                println(Math.ceil(n_expected).
                    toInteger())
                break
            }
            n_max++
        }
    }
```

Appendix B

PRISM Code for CTMC Model of DNS BAA

Model source code when AGF is used:

```
// A model for the DNS bandwidth amplification (BAA) attack
// with ''AGF'' as the countermeasure
ctmc

// bogus rate for each zombie machine
const bogus_rate=10;

// number of zombies participating in the attack
const zombies;

// base arrival rate for legitimate packets
const Rl=100;

// detection fraction for filtering
const double df = 0.9;

// false-positive fraction for filtering
const double fpf = 0.1;

// number of aggressive retries, starts from 0
const retries;

// increased rate of legitimate packets based on retries
const actual_Rl = ceil(pow(2, retries));

// amplification factor
const double AF;
```

```
// Bandwidth size (458 for DNS and 112 for DNSSec)
const BW;

// rate at which the CS processes the queued packets
const serve_rate = 12666;

// A successful BAA attack is experienced when for the
// victim, BW_queue = BW.  This means that the victim's
// bandwidth is full and it can no longer accept incoming
// packets.  Any incoming packets are dropped till
// bandwidth is freed.
formula BandwidthExpired = (BW_queue=BW);
formula DenialOfService = (legitimateRequestInitiated=false
   ) & BandwidthExpired;

// Implements the packet filter mechanism
module Filter

[receive_bogus_packet] true -> (1-df):true;

[receive_legit_packet] true-> (1-fpf):true;

[client_request] true-> (1-fpf):true;

endmodule

// The DNS server for the client machine.  The CS is
// the victim of the BAA attack.
module CS

// Queue for all incoming packets.  Growth and shrinkage of
// this queue represents the way CS's bandwidth is consumed
// and freed up
BW_queue : [0..BW] init 0;

// The BW_queue is incremented by 1 with Rl when a
// legitimate packet is received.
[receive_legit_packet] (BandwidthExpired=false)
-> actual_Rl:(BW_queue'=BW_queue+1);

// The legitimate packets that are not serviced due to BAA
[legit_packet_lost] BandwidthExpired=true
-> actual_Rl : true;

// When bogus traffic arrives, the BW_queue is incremented
```

```
// by 1 with the rate determined by AF.  The effective rate
// of this action is bogus_rate*zombies*AF
[receive_bogus_packet] BandwidthExpired=false
-> AF:(BW_queue'=BW_queue+1);

// Bogus traffic that arrives after bandwidth expiration
[bogus_packet_lost] BandwidthExpired=true
-> AF:true;

// Serve (outgoing) traffic
// The BW_queue is decremented by 1 with serve_rate.
[serve_queued_packet] BW_queue > 0
-> serve_rate : (BW_queue'=BW_queue-1);

// A single action from client which is used to compute the
// probability of failure.  The BW_queue is incremented by
// 1 with rate Rl when a legitimate packet is received
[client_request] (BandwidthExpired=false)
-> (BW_queue'=BW_queue+1);

endmodule


// Represents the ''rest of the internet'' excluding CS
module Net

legitimateRequestInitiated : bool init false;

[receive_bogus_packet] true ->bogus_rate*zombies:true;

[bogus_packet_lost] true ->bogus_rate*zombies:true;

[receive_legit_packet] (BandwidthExpired=false)
-> 1 : true;

[legit_packet_lost] (BandwidthExpired=true)
-> 1 : true;

// A single action from client which is used to compute the
// probability of failure.
[client_request] (legitimateRequestInitiated = false)
-> actual_Rl:(legitimateRequestInitiated'= true);

endmodule
```

```
// Reward definitions

rewards ``R1''
[receive_legit_packet] true : 1;
[client_request] true : 1;
endrewards

rewards ``R2''
[receive_bogus_packet] true : 1;
endrewards

rewards ``R3''
BW_queue < BW : 1;
endrewards
```

Properties file:

```
// Time-bound for evaluating cumulative reward-based
// properties.  For our experiments, we set t to 0.1
const double t;

// Probability that the client is not serviced due to BAA
P=? [ F DenialOfService ]

// Cumulative reward-based property P1
R{``R1''}=? [ C<=t ]

// Cumulative reward-based property P2
R{``R2''}=? [ C<=t ]

// Cumulative reward-based property P3
R{``R3''}=? [ C<=t ]
```

Appendix C

PRISM Code for Stochastic Game-Based Model of DNS BAA

Model source code when AGF is used:

```
// Game-based model of DNS BAA.
smg

// Maximum bandwidth available.
const int BW = 458;

// Maximum time for which the model should run.
const int maxTime = 23;

// Rate at which a single zombie sends bogus packets
const double bogus_rate = 10 * AF;

// Amplification factor
const double AF = 15.31;

// Legitimate packet rate
const double Rl = 100;

// Total rate at which bogus packets arrive
formula Rb = bogus_rate * zombies;

// Detection fraction
const double df = 0.9;

// False-positive fraction
const double fpf = 0.1;

// Random drop fraction
```

```
const double rdf;

// Number of retries
const int retries;

const int MAX_SUCC_ATTACKS;
const int ATTACKER_LATENCY;

// Defines the time for which the defender must wait before
// it can re-enable the countermeasures after having
// disabled them once.  This is typically very low and
// thus it is set to zero
const int DEFENDER_LATENCY = 0;

// For brevity, we define CS's probabilities of receiving
// packets when either FTR or AGR used.  Such probabilities
// can be defined in a similar way if defender uses
// Nofix, RND, RDR, or AGF.

// CS's probabilities of receiving packets if FTR is used
formula bw_factor_FTR = (BW < (Rb_in_FTR + Rl_in_FTR))?(BW
   /(Rb_in_FTR + Rl_in_FTR)):1;
formula Rb_in_FTR = Rb * (1 - df);
formula Rl_in_FTR = Rl * (1 - fpf);
formula Rb_drop_FTR = Rb * df;
formula Rl_drop_FTR = Rl * fpf;
formula p_b_receive_FTR = (Rb_in_FTR / (Rb + Rl)) * (
   bw_factor_FTR);
formula p_l_receive_FTR = (Rl_in_FTR / (Rb + Rl)) * (
   bw_factor_FTR);
formula p_b_drop_FTR = (Rb_drop_FTR / (Rb + Rl)) + (
   Rb_in_FTR / (Rb + Rl)) * (1 - bw_factor_FTR);
formula p_l_drop_FTR = 1 - (p_b_receive_FTR +
   p_l_receive_FTR + p_b_drop_FTR);

// CS's probabilities of receiving packets if AGR is used
formula bw_factor_AGR = (BW < (Rb_in_AGR + Rl_in_AGR))?(BW
   /(Rb_in_AGR + Rl_in_AGR)):1;
formula Rl_AGR = ceil(Rl * pow(2, retries));
formula Rb_in_AGR = Rb;
formula Rl_in_AGR = Rl_AGR;
formula Rb_drop_AGR = 0;
formula Rl_drop_AGR = 0;
formula p_b_receive_AGR = (Rb_in_AGR / (Rb + Rl_AGR)) * (
   bw_factor_AGR);
```

```
formula p_l_receive_AGR = (Rl_in_AGR / (Rb + Rl_AGR)) * (
   bw_factor_AGR);
formula p_b_drop_AGR = (Rb_drop_AGR / (Rb + Rl_AGR)) + (
   Rb_in_AGR / (Rb + Rl_AGR)) * (1 - bw_factor_AGR);
formula p_l_drop_AGR = 1 - (p_b_receive_AGR +
   p_l_receive_AGR + p_b_drop_AGR);

// Keep track of the elapsed time.
global time : [0..maxTime] init 0;

// Scheduling variable.  Initialized to two to enable
// the defender.
global sched : [0..2] init 2;

global baseDistEnabled : bool init false;
global ftrDistEnabled : bool init false;
global agrDistEnabled : bool init false;
global rndDistEnabled : bool init false;
global rndAgrDistEnabled : bool init false;
global ftrAgrDistEnabled : bool init false;

// Player definitions
player DFD
        Defender
endplayer

player CS
        ClientServer
endplayer

player AS
        Attacker
endplayer


// Defender nondeterministically chooses a countermeasure
module Defender

disableCM : bool init false;
defenderLatency : [-1..DEFENDER_LATENCY] init
   DEFENDER_LATENCY;
readyToChooseDefense : bool init false;

// Turn the countermeasure on/off
```

```
[] sched = 1 & !readyToChooseDefense & defenderLatency =
   DEFENDER_LATENCY
-> (disableCM'=false) & (readyToChooseDefense'=true);

[] sched = 1 & !readyToChooseDefense & !disableCM
-> (disableCM'=true) & (defenderLatency'=-1);

[] sched = 1 & disableCM & defenderLatency <
   DEFENDER_LATENCY & !readyToChooseDefense
-> (defenderLatency'=defenderLatency + 1) & (
   readyToChooseDefense'=true);

// Enable NoFix
[] sched = 1 & time < maxTime & disableCM &
   readyToChooseDefense & defenderLatency <
   DEFENDER_LATENCY
-> (baseDistEnabled'=true) & (ftrDistEnabled'=false) & (
   agrDistEnabled'=false) & (rndDistEnabled'=false) & (
   ftrAgrDistEnabled'=false) & (rndAgrDistEnabled'=false) &
    (readyToChooseDefense'=false) & (sched'= 0);

[] sched = 1 & time < maxTime & disableCM &
   readyToChooseDefense & defenderLatency =
   DEFENDER_LATENCY
-> (disableCM'=false) & (baseDistEnabled'=true) & (
   ftrDistEnabled'=false) & (agrDistEnabled'=false) & (
   rndDistEnabled'=false) & (ftrAgrDistEnabled'=false) & (
   rndAgrDistEnabled'=false) & (readyToChooseDefense'=false
   ) & (sched'= 0);

// Enable FTR
[] sched = 1 & time < maxTime & !disableCM &
   readyToChooseDefense
-> (baseDistEnabled'=false) & (ftrDistEnabled'=true) & (
   agrDistEnabled'=false) & (rndDistEnabled'=false) & (
   ftrAgrDistEnabled'=false) & (rndAgrDistEnabled'=false) &
    (readyToChooseDefense'=false) & (sched'= 0);

// Enable AGR
[] sched = 1 & time < maxTime & !disableCM &
   readyToChooseDefense
-> (baseDistEnabled'=false) & (ftrDistEnabled'=false) & (
   agrDistEnabled'=true) & (rndDistEnabled'=false) & (
   ftrAgrDistEnabled'=false) & (rndAgrDistEnabled'=false) &
    (readyToChooseDefense'=false) & (sched'= 0);
```

```
[] sched = 1 & time = maxTime -> (sched'= 0);

endmodule

// ClientServer (CS) is the victim DNS server.
module ClientServer

readyToScheduleDefenderAndAttacker : bool init false;
legitPacketReceived : bool init false;
bogusPacketReceived : bool init false;
legitPacketDropped : bool init false;
bogusPacketDropped : bool init false;

// Handle FTR
[] sched = 0 & ftrDistEnabled & time < maxTime& !
   readyToScheduleDefenderAndAttacker
-> p_b_receive_FTR : (legitPacketReceived'= false) & (
   bogusPacketReceived'= true) & (legitPacketDropped'=
   false) & (bogusPacketDropped'= false) & (
   readyToScheduleDefenderAndAttacker'=true) +
p_l_receive_FTR : (legitPacketReceived'= true) & (
   bogusPacketReceived'= false) & (legitPacketDropped'=
   false) & (bogusPacketDropped'= false) & (
   readyToScheduleDefenderAndAttacker'=true) +
p_b_drop_FTR : (legitPacketReceived'= false) & (
   bogusPacketReceived'= false) & (legitPacketDropped'=
   false) & (bogusPacketDropped'= true) & (
   readyToScheduleDefenderAndAttacker'=true) +
p_l_drop_FTR : (legitPacketReceived'= false) & (
   bogusPacketReceived'= false) & (legitPacketDropped'=
   true) & (bogusPacketDropped'= false) & (
   readyToScheduleDefenderAndAttacker'=true);

// Handle AGR
[] sched = 0 & agrDistEnabled & time < maxTime& !
   readyToScheduleDefenderAndAttacker
-> p_b_receive_AGR : (legitPacketReceived'= false) & (
   bogusPacketReceived'= true) & (legitPacketDropped'=
   false) & (bogusPacketDropped'= false) & (
   readyToScheduleDefenderAndAttacker'=true) +
p_l_receive_AGR : (legitPacketReceived'= true) & (
   bogusPacketReceived'= false) & (legitPacketDropped'=
   false) & (bogusPacketDropped'= false) & (
   readyToScheduleDefenderAndAttacker'=true) +
```

```
p_b_drop_AGR : (legitPacketReceived'= false) & (
   bogusPacketReceived'= false) & (legitPacketDropped'=
   false) & (bogusPacketDropped'= true) & (
   readyToScheduleDefenderAndAttacker'=true) +
p_l_drop_AGR : (legitPacketReceived'= false) & (
   bogusPacketReceived'= false) & (legitPacketDropped'=
   true) & (bogusPacketDropped'= false) & (
   readyToScheduleDefenderAndAttacker'=true);

// Enable the attacker
[] sched = 0 & time < maxTime &
   readyToScheduleDefenderAndAttacker
-> 1 : (sched'= 2) & (time'= time + 1) & (
   legitPacketReceived'= false) & (bogusPacketReceived'=
   false) & (legitPacketDropped'= false) & (
   bogusPacketDropped'= false) & (
   readyToScheduleDefenderAndAttacker'=false);

[] sched = 0 & time = maxTime -> true;

endmodule


// The attacker sends large volumes of packets to the
// victim DNS server.
module Attacker
zombies : [0..100] init 0;

succAttacks : [0..MAX_SUCC_ATTACKS] init 0;
attackLatency : [0..ATTACKER_LATENCY] init 0;
attackEnabled : bool init false;

[] sched = 2 & time <= maxTime & !attackEnabled &
   succAttacks = MAX_SUCC_ATTACKS & attackLatency <
   ATTACKER_LATENCY
-> (attackEnabled'=false) & (attackLatency'=attackLatency
   +1) & (zombies'=0) & (sched'=1);

[] sched = 2 & time <= maxTime & !attackEnabled &
   succAttacks = MAX_SUCC_ATTACKS & attackLatency =
   ATTACKER_LATENCY
-> (attackEnabled'=false) & (succAttacks'=0) & (
   attackLatency'=0);
```

```
[] sched = 2 & time <= maxTime & !attackEnabled &
   succAttacks < MAX_SUCC_ATTACKS
-> (attackEnabled'=false) & (zombies'=0) & (sched'=1) & (
   succAttacks'=0);

[] sched = 2 & time <= maxTime & !attackEnabled &
   succAttacks < MAX_SUCC_ATTACKS
-> (attackEnabled'=true) & (succAttacks'=succAttacks + 1);

// Nondeterministically choose number of zombies to use to
// launch the DNS BAA
[] sched = 2 & time <= maxTime & attackEnabled -> (zombies
   '=1) & (attackEnabled'=false) & (sched'=1);
[] sched = 2 & time <= maxTime & attackEnabled -> (zombies
   '=100) & (attackEnabled'=false) & (sched'=1);
[] sched = 2 & time = maxTime -> (sched'=1);

endmodule

rewards ``AttackerPayoff1''
sched = 0 & legitPacketDropped & zombies > 0 : 1/zombies;
sched = 0 & legitPacketReceived & zombies > 0 : -1/zombies;
sched = 0 & legitPacketDropped & zombies = 0 : 1;
sched = 0 & legitPacketReceived & zombies = 0 : -1;
endrewards

rewards ``AttackerPayoff2''
sched = 0 & legitPacketReceived : -1;
sched = 0 & bogusPacketReceived : 1;
endrewards
```

Properties file:

```
// Generate optimal attack strategy that would maximize
// attacker's payoff accumulated over the duration of the
// experiment
<<AS>> R{``AttackerPayoff1''}max=? [ F time=maxTime ]
<<AS>> R{``AttackerPayoff2''}max=? [ F time=maxTime ]
```