Formal Modeling and Analysis of DoS Using Probabilistic Rewrite Theories*

Gul Agha, Michael Greenwald, Carl A. Gunter, Sanjeev Khanna
Jose Meseguer, Koushik Sen, and Prasannaa Thati†

Abstract

Existing models for analyzing the integrity and confidentiality of protocols need to be extended to enable the analysis of availability. Prior work on such extensions shows promising applications to the development of new DoS countermeasures. Ideally, it should be possible to apply these countermeasures systematically in a way that preserves desirable properties already established. This paper investigates a step toward achieving this ideal by describing a way to expand term rewriting theories to include probabilistic aspects that can be used to show the effectiveness of DoS countermeasures. In particular, we consider the shared channel model, in which adversaries and valid participants share communication bandwidth according to a probabilistic interleaving model, and a countermeasure known as selective verification applied to the handshake steps of the TCP reliable transport protocol. These concepts are formulated in a probabilistic extension of the Maude term rewriting system, called PMAUDE. Furthermore, we formally verified the desired properties of the countermeasures through automatic statistical model-checking techniques.

1 Introduction

There are well-understood models on which to base the analysis of integrity and confidentiality. The most common approaches are algebraic techniques [6] based on idealized cryptographic primitives and complexity-theoretic techniques [5] based on assumptions about complexity. There has also been progress on unified perspectives that enable using the simpler algebraic techniques to prove properties like those ensured by the more complete cryptographic techniques. However, neither of these approaches or their unifications are designed to approach the problem of availability threats in the protocols they analyze. For example, suppose a protocol begins with a sender sending a short message to a receiver, where the receiver's first step is to verify a public key signature on the message. A protocol like this is generally considered to be problematic because an adversarial sender can send many packets with bad signatures at little cost to himself while the receiver will need to work hard to (fail to) verify these signatures. Algebraic and complexity-theoretic analysis techniques ensure only that the recipient will not be fooled by the bad packets and will not leak information as a result of receiving them. However, they do not show that the receiver will be available to a valid sender in the presence of one or more attackers.

In [7] we began an effort to explore a formal model for the analysis of DoS based on a simple probabilistic model called the "shared channel" model. This effort showed that the shared channel model could be used to prove properties of DoS countermeasures for authenticated broadcast that could be verified in experiments. We have subsequently conducted a number of experiments to explore the application of such countermeasures to other classes of protocols. The aim of this paper is to explore the prospects for using the shared channel model as a

*This work was supported in part by ONR Contract N00014-02-1-0715.

†Addresses of the authors: K. Sen, G. Agha, C. A. Gunter, J. Meseguer, University of Illinois at Urbana-Champaign; Michael Greenwald, Lucent Bell Labs; Sanjeev Khanna, University of Pennsylvania; Prasannaa Thati, Carnegie-Mellon University,

foundation for extending term rewriting models of network protocols to cover DoS aspects of the protocols and their modification with counter-measures. Our particular study is to investigate the use of a probabilistic extension of the Maude rewrite system called PMAUDE and its application to understanding the effectiveness of a DoS countermeasure known as "selective sequential verification" [7]. This technique was explored for authenticated broadcast in [7] but in the current paper we consider its application to handshake steps of the TCP reliable transport protocol.

At a high level, our ultimate aim is to demonstrate techniques for showing how a network protocol can be systematically "hardened" against DoS using probabilistic techniques while preserving the underlying correctness properties the protocol was previously meant to satisfy. Specifically, given a protocol P and a set of properties T, we would like to expand T to a theory T^* that is able to express availability properties and show that a transformation P^* of P meets the constraints in T^* without needing to re-prove the properties T that P satisfied in the restricted language. The shared channel model provides a mathematical framework for this extension.

In this paper, we develop a key element of this program: a formal language in which to express the properties T^* and show that availability implications hold for P^*. We attempt to validate this effort by showing its effectiveness on a selective verification for TCP. In particular, we show how we can specify TCP/IP 3-way handshake protocol in PMAUDE algebraically. First, we take a previously specified formal non-deterministic model of the protocol. We then replace all non-determinism by probabilities. The resulting model with quantified non-determinism (or probabilities) is then analyzed for quantitative properties such as availability. The analysis is done by combining Monte-Carlo simulation of the model with statistical reasoning. In this way, we leverage the existing modelling and reasoning techniques to quantified reasoning without interfering with the underlying non-quantified properties of the model.

The rest of the paper is organized as follows. In Section 2, we give the preliminaries of DoS theory followed by its application to TCP/IP 3-way handshaking protocol in Section 3. Then we briefly describe PMAUDE in Section 4. In Section 5, we describe and discuss the algebraic probabilistic specification of DoS hardened TCP/IP protocol in PMAUDE. We describe the results of our analysis of some desired properties written in the query language for the specification of TCP/IP protocol in Section 6.

2 DoS Theory

On the face of it, the conventional techniques for establishing confidentiality and integrity are inappropriate for analyzing DoS, since they rely on very strong models of the adversary's control of the network. In particular, they assume that the adversary is able to delete packets from the network at will. An adversary with this ability has an assured availability attack. Typical analysis techniques therefore adapt this assumption in one of two ways. A first form of availability analysis is to focus on the relationship between the sender and the attacker and ask whether the attacker/sender is being forced to expend at least as much effort as the valid receiver. In our example, this is an extremely disproportionate level of effort, since forming a bad signature is much easier than checking that it is bad. Thus the protocol is vulnerable to the imposition of a disproportionate effort by the receiver. This is a meaningful analysis, but it does not answer the question of whether a valid sender will experience the desired availability. A second form of availability analysis is to ask whether the receiver can handle a specified load. For instance, a stock PC can check about 8000 RSA signatures each second, and it can receive about 9000 packets (1500 bytes per packet) each second over a 100Mbps link. Thus a receiver is unable to check all of the signatures it receives over such a channel. A protocol of the kind we have envisioned is therefore deemed to be vulnerable to a *signature flood* attack based on cycle exhaustion. By contrast, a stock PC can check the hashes on 77,000 packets each second, so a receiver that authenticates with hashes can service all of its bandwidth using a fraction of its capacity. This sort of analysis leads one to conclude that a protocol based on public key signatures is vulnerable to DoS while one based on hashes is not.

These techniques are sound but overly conservative, because they do not explicitly account for the significance of *valid* packets that reach the receiver. Newer techniques for analyzing DoS have emerged in the last year that provide a fresh perspective by accounting for this issue. In essence, these new models are both more realistic for the Internet and suggest new ideas for countermeasures. We refer to one basic version of this new approach as the *shared channel model*. The shared channel model is a four-tuple consisting of the minimum bandwidth W_0 of the sender, the maximum bandwidth W_1 of the sender (where $W_0 \leq W_1$), the bandwidth α of the adversary, and the loss rate p of the sender where $0 \leq p < 1$. The ratio $R = \alpha/W_1$ is the *attack factor* of the model. When $R = 1$, this is a *proportionate* attack and, when $R > 1$, it is a *disproportionate* attack. As in the algebraic model, the adversary is assumed to be able to replay packets seen from valid parties and flood the target with anything he can form from these. But in the shared channel model he is not able to delete specific packets from the network. In effect, he is able to interleave packets among the valid ones at a specified maximum rate. This interleaving may contribute to the loss rate p of the sender, but the rate of loss is assumed to be bounded by p and randomly applied to the packets of the sender.

The key insight that underlies the techniques in this paper arises from recognizing the *asymmetry* the attacker aims to exploit; his willingness to spend his entire bandwidth on an operation that entails high cost for the receiver also offers opportunities to burden the attacker in disproportionate ways relative to the valid sender. This can be seen in a simple strategy we call *selective verification.* The idea is to cause the receiver to treat the signature packets she receives as arriving in an *artificially* lossy channel. The sender compensates by sending extra copies of his signature packets. If the recipient checks the signature packets she receives with a given probability, then the number of copies and the probability of verification can be varied to match the load that the recipient is able to check. For example, suppose a sender sends a 10Mbps stream to a receiver, but this is mixed with a 10Mbps stream of DoS packets devoted entirely to bad signatures. To relieve the recipient of the need to check all of these bad signatures, the receiver can check signatures with a probability of 25%, and, if the sender sends about 20 copies of each signature packet, the receiver will find a valid packet with a probability of more than 99% even if the network drops 40% of the sender's packets. This technique is inexpensive, scales to severe DoS attacks, and is adaptable to many different network characteristics.

3 SYN Floods as DoS for TCP/IP

TCP is an extremely common reliable bi-directional stream protocol that uses a three-way handshake to establish connections. Glossing over many details, a sender initiates a connection by sending a packet with the SYN flag set and an initial sequence number. The receiver responds by acknowledging the SYN flag, and sending back a SYN with its *own* sequence number. When the original sender acknowledges the receiver's SYN (this ACK is the 3rd packet in a 3-way handshake), then the connection is ESTABLISHED.

Each established connection requires a TCB (Transmission Control Block) at each end of the connection. The TCB occupies a few hundred bytes of identification and control information, statistics, as well as a much larger allocation of packet buffers for received data and (re)transmission queues. In most operating system kernels, both packet buffer space and the number of available TCBs are fixed at boot time, and they constitute a limited resource. This opens a significant vulnerability to adversaries who aim to overwhelm this limit by flooding a server with SYN packets; this is typically called a *SYN flood attack.* This threat is mitigated in many systems by storing connection information in a SYN cache (a lighter-weight data structure, recording only identity information and sequence numbers for the connection) until the connection becomes ESTABLISHED, at which point the (more expensive) full TCB is allocated. Normally, a legitimate connection occupies a slot in the SYN cache for only one round trip time (RTT). If no ACK for the SYN+ACK arrives, then the server eventually removes the entry from the SYN cache, but only after a much longer timeout interval, t_A.

SYN flooding constitutes an easy denial of service attack because SYN cache entries are relatively scarce, while the bandwidth needed to send a single SYN packet is relatively cheap. The attacker gains further leverage from the disparity between the one RTT slot occupancy (often on the order of a millisecond or less) for a legitimate client, compared with a fraudulent SYN packet that typically holds a syn-cache slot for a value of t_A ranging from 30-120 seconds.

A SYN attack is simple to model; attackers merely send SYN packets at a cumulative rate which we denote by r_A. We can compute the effectiveness of the DoS attack by the probability of success of a client's attempt to connect, and from that compute the number of legitimate connections per second that the server can support under a given attack rate r_A. If the server offers no defense, and if the order in which incoming SYNs are processed at the server is adversarially chosen, then it is clear that an attack rate r_A of $O(B/t_A)$ suffices to completely take over a syn-cache of size B. To see this, observe that in every second, B/t_A of the attacker's slots in the SYN cache expire, and B/t_A new ones arrive to take their places. Even in a more realistic model where the incoming SYN requests are assumed to be ordered in accordance with a random permutation, it is easy to show that an attack rate of $O(B/t_A)$ suffices.

It is clear from this analysis (as well as from abundant empirical evidence) that even a moderate rate of DoS attack can totally disable a server. For a server with a SYN cache of size $B = 10,000$ and a timeout interval of 75 seconds a moderate attack rate of 200 to 300 SYNS per second is enough to almost completely overwhelm the server! (An energetic attacker can generate SYN packets 1000 times as quickly as this on a commodity 100Mbps Fast Ethernet link.)

Selective verification can improve this performance significantly[1]. Let B denote the number of slots in the SYN cache. Suppose we want to ensure that the attacker never blocks more than a fraction f of the table, for $0 < f < 1$. We ask the server to process each incoming SYN with probability p where p satisfies $pt_A r_A \leq fB$, then we ensure that at least a $(1 - f)$-fraction of the SYN cache is available to legitimate users. We effectively inflate the bandwidth cost of mounting an attack rate of r_A to be r_A/p. Considering once again an attacker on 100 Mbps channel ($300,000$ SYNs/sec), if we set $p = 10^{-3}/6$, we ensure that the attacker cannot occupy more than half the table at any point in time. The attacker can still deny service, but is now required to invest as much in bandwidth resources as the collective investment of the clients that it is attacking.

If we increase the cache size by a factor of 30, we can get an identical guarantee with $p = .005$. The overhead on a valid client to establish a connection then is only 200 SYN packets, roughly 8KB, for each request. These overheads are not insignificant but they allow us to provide unconditional guarantees on availability of resources for valid clients. If we downloaded the PS version of this paper (500KB), the blowup increases the transfer size by 2%. Moreover, these overheads should be contrasted with the naive alternative: the cache size would have to be increased to 6×10^7 to get the same guarantee.

4 Probabilistic Rewrite Theories

Rewriting logic is an expressive semantic framework to specify a wide range of concurrent systems [11]. In practice, however, some systems may be probabilistic in nature, either because of their environment, or by involving probabilistic algorithms by design, or both. This raises the question of whether such systems can also be formally specified by means of rewrite rules in some suitable probabilistic extension of rewriting logic. This would provide a general formal specification framework for probabilistic systems and could support different forms of symbolic simulation and formal analysis. In particular, DoS-resistant communication protocols such as the DoS-hardened TCP/IP protocol discussed in Section 3 could be formally specified and analyzed this way.

The notion of a probabilistic rewrite theory provides an example of such a semantic framework. Usually, the rewrite rules specifying a non-probabilistic system are of the form

[1]Techniques such as SYN cookies are also effective against SYN flooding; however they do not preserve the underlying behavior of TCP.

$$t \Rightarrow t' \quad \texttt{if} \; C$$

where the variables appearing in t' are typically a subset of those appearing in t, and where C is a condition. The intended meaning of such a rule is that if a fragment of the system's state is a substitution instance of the pattern t, say with substitution θ, and the condition $\theta(C)$ holds, then our system can perform a local transition in that state fragment changing it to a new local state $\theta(t')$. Instead, in the case of a probabilistic system, we will be using rewrite rules of the form,

$$t(\overrightarrow{x}) \Rightarrow t'(\overrightarrow{x}, \overrightarrow{y}) \;\texttt{if}\; C(\overrightarrow{x}) \;\texttt{with probability}\; \overrightarrow{y} := \pi_r(\overrightarrow{x})$$

where the first thing to observe is that the term t' has new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ appearing in t. Therefore, such a rule is *non-deterministic*; that is, the fact that we have a matching substitution θ such that $\theta(C)$ holds, does not uniquely determine the next state fragment: there can be many different choices for the next state depending on how we instantiate the extra variables $\overrightarrow{y}$. In fact, we can denote the different such next states by expressions of the form $t'(\theta(\overrightarrow{x}), \rho(\overrightarrow{y}))$, where θ is fixed as the given matching substitution, but ρ ranges along all the possible substitutions for the new variables $\overrightarrow{y}$. The probabilistic nature of the rule is expressed by the notation `with probability` $\overrightarrow{y} := \pi_r(\overrightarrow{x})$, where $\pi_r(\overrightarrow{x})$ is a probability distribution *which depends on the matching substitution* θ, and we then choose the values for $\overrightarrow{y}$, that is the substitution ρ, probabilistically according to the distribution $\pi_r(\theta(\overrightarrow{x}))$.

We can illustrate these ideas with a very simple example, namely a digital battery-operated clock that measures time in seconds. The state of the clock is represented by a term `clock(t,c)`, where `t` is the current time in seconds, and `c` is a rational number indicating the amount of charge in the battery. The clock ticks according to the following probabilistic rewrite rule:

$$\texttt{clock(t,c)} \Rightarrow \texttt{if B then clock(t + 1,c}-\tfrac{\texttt{c}}{1000}\texttt{) else broken(t,c} - \tfrac{\texttt{c}}{1000}\texttt{) fi}$$
$$\texttt{with probability B := BERNOULLI(}\tfrac{\texttt{c}}{1000}\texttt{)} \;.$$

Note that the rule's righthand side has a new boolean variable `B`. If all goes well (`B = true`), then the clock increments its time by one second and the charge is slightly decreased; but if `B = false`, then the clock will go into a broken state `broken(t,c -` $\frac{\texttt{c}}{1000}$`)`. Here the boolean variable `B` is distributed according to the Bernoulli distribution with mean $\frac{\texttt{c}}{1000}$. Thus, the value of `B` *probabilistically depends on the amount of charge* left in the battery: the lesser the charge level, the greater the chance that the clock will break; that is, we have different probability distributions for different matching substitutions θ of the rule's variables (in particular, of the variable `c`).

Of course, in this example the variable `B` is a discrete binary variable; but we could easily modify this example to involve continuous variables. For example, we could have assumed that `t` was a real number, and we could have specified that the time is advanced to a new time `t + t'`, with `t'` a new real-valued variable chosen according to an exponential distribution. In general, the set of new variables $\overrightarrow{y}$ could contain both discrete and continuous variables, ranging over different data types. In particular, both discrete and continuous time Markov chains can be easily modeled, as well as a wide range of discrete or continuous probabilistic systems, which may also involve nondeterministic aspects [9]. Furthermore, the PMAUDE extension of the Maude rewriting logic language allows us to symbolically simulate probabilistic rewrite theories [10, 3], and we can formally analyze their properties according to the methods described in [3]. Due to space constraints, we do not give the mathematical definition of probabilistic rewrite theories. Readers are referred to [10, 9] for such details.

In general, a probabilistic rewrite theory $\mathcal{R}$ *involves both probabilities and non-determinism.* The non-determinism is due to the fact that, in general, *different rules, possibly with different subterm positions and substitutions* could be applied to rewrite a given state u: the choice of what rule to apply, and where, and with which substitution is *non-deterministic*. It is only when such a choice has been made that probabilities come into the picture, namely for choosing the substitution ρ for the new variables $\overrightarrow{y}$. In particular, for the kind of statistical

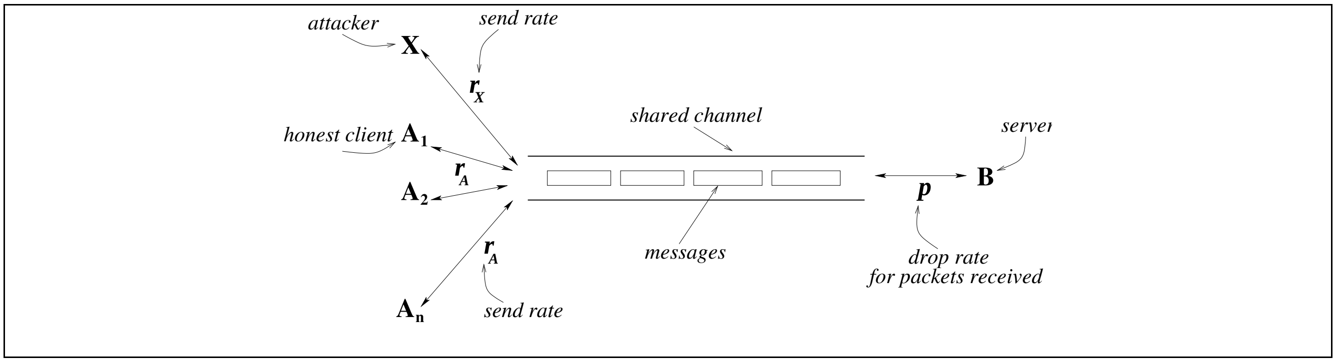

Figure 1: An instance of the TCP's 3-way handshake protocol.

model checking discussed in [13, 14] that will be used to formally analyze our DoS-resistant TCP/IP protocol, we need to assume that *all non-determinism has been eliminated* from our specification; that is, that at most one single rule, position, and substitution are possible to rewrite any given state.

What this amounts to, in the specification of a concurrent system such as a network protocol, is the *quantification of all non-determinism due to concurrency using probabilities*. This is natural for simulation purposes and can be accomplished by requiring the probabilistic rewrite theory to satisfy some simple requirements described in [3].

We will consider rewrite theories specifying concurrent actor-like objects [2] and communication by asynchronous message passing; this is particularly appropriate for communication protocols. In rewriting logic, such systems (see [12] for a detailed exposition) have a distributed state that can be represented as a *multiset* of objects and messages, where we can assume that objects have a general record-like representation of the form: $\langle \texttt{name:}\ o \mid a_1 : v_1, \ldots a_1 : v_1\rangle$, where o is the object's name and the $a_i : v_i$ its corresponding attribute-value pairs in a given state. It is also easy to model in this way *real-time concurrent object systems*: one very simple way to model them is to include a global clock as a special object in the multiset of objects and messages. Rewrite rules in such a system will involve an object, a message, and the global time and will consume the message, change the object's state, and send messages to other objects. To deal with message delays and their probabilistic treatment, we can represent messages as *scheduled objects* that are inactive until their associated delay has elapsed.

5 Probabilistic Rewrite Specification of DoS resistant TCP 3-way Handshaking

We now present an executable specification of TCP's 3-way handshake protocol in probabilistic rewriting logic. We consider a protocol instance composed of N honest clients $C_1, \ldots, C_N$ trying to establish a TCP connection with the server S, and a single attacker A that launches a SYN-flood attack on S (see Figure 1). The clients C_i transmit SYN requests to S at the rate r_C, while the attacker A floods spurious SYN requests at the rate r_A. These rates are assumed to be parameters of an exponential distribution from which the time for sending the next packet is sampled. The server S drops each packet it receives, independently, with probability p. We assume that each message across the network is subject to a constant transmission delay d. Of course, these assumptions about the various distributions can be easily changed in the implementation that follows.

Each client C_i is modeled as an object with four attributes as follows.

```
<name: C(i) | isn:N, repcnt:s(CNT), sendto:SN, connected:false>
```

The attribute `isn` specifies the sequence number that is to be used for the TCP connection, `sendto` specifies the name of server S, `repcnt` specifies the number of times the SYN request is to be (re)transmitted in order to account for random dropping of packets at S, and `connected` specifies if the connection has been successfully established as yet. The attacker is modeled as an object with a single attribute as follows.

```
<name: AN | sendto: SN >
```

The server S is modeled as an object with two attributes.

```
<name: SN | isn: M , synlist: SC >
```

The attribute `isn` specifies the sequence number that S uses for the next connection request it receives, while `synlist` is the SYN cache that S maintains for the pending connection requests.

Following is the probabilistic rewrite rule that models the client C_i sending a SYN request.

```
<name:C(i) | isn:N, repcnt:s(CNT), sendto:SN, connected:false> (C(i)← poll) T
            ⇒ <name: C(i) | isn:N, repcnt:CNT, sendto:SN, connected:false>
              [ d + T , (SN← SYN(C(i),N)) ] [ t + T , (C(i)← poll) ]   T
                                  with probability t := EXPONENTIAL(r_C)   .
```

We use special poll messages to control the rate at which C_i retransmits the SYN requests. Specifically, C_i repeatedly sends itself a poll message, and each time it receives a poll message it sends out a SYN request to S. The poll messages are subject to a random delay t that is sampled from the exponential distribution with parameter r_C. Specifically, the message is scheduled at time $t + T$, where T is the current global time. The net effect of this is that C_i sends SYN requests to S at rate r_C. Perhaps it is important to point out that the poll messages are not regular messages that are transmitted across the network; they have been introduced only for modeling purposes. Further, note that the approach of simply freezing C_i by scheduling it at time $T + t$ does not work since that would also prevent C_i from receiving any SYN+ACK messages that it may receive from S meanwhile. Finally, note that the replication count is decremented by one after the transmission of SYN message, and the message itself is scheduled with a delay d.

The rule for SYN flooding by the attacker is very similar, except that it uses randomly generated sequence numbers.

```
<name: AN |  sendto: SN > (AN ←  poll) T ⇒ <name: AN | sendto: SN >  T
          [ d + T , (SN← SYN(AN,random(counter))) ] [ t + T , (AN← poll) ]
                                   with probability t := EXPONENTIAL(r_A)   .
```

The following rule models the processing of SYN requests by the server S.

```
<name: SN | isn: M , synlist: SC > (SN← SYN(ANY,N)) T
⇒ if(drop? or size(SC) > SYN-CACHE-SIZE) then <name: SN | isn: M,synlist: SC > T
  else <name: SN | isn:s(M), synlist:add(SC,entry(ANY,M))>
      [d+T,(ANY← SYN+ACK(SN,N,M))] [TIMEOUT+T,(SN← tmout(entry(ANY,M)))] T fi
                                  with probability drop? := BERNOULLI(p)   .
```

The random dropping of incoming messages is modeled by sampling from the Bernoulli distribution with the appropriate parameter p. An incoming request can also be dropped if the SYN cache is full. If the cache is not full, for each request that is not dropped, the server S makes an entry for the request in the cache, and sends out a SYN+ACK message to the source of the request. A cache entry is of the form `entry(N,M)` where `N` is the name of the source which has requested a connection, and `M` is the sequence number for the connection. Timing out of entries in the cache is modeled by locally sending a message to self that is scheduled after an interval of time equal to the timeout period. Here is the rule for removing timed out entries.

```
<name: SN|isn: N,synlist: [s(SZ),(L1 entry(ANY,M) L2)]> (SN ←
tmout(entry(ANY,M)))
     ⇒  <name: SN | isn: N , synlist: [ SZ , (L1 L2) ] >   .
```

The first argument in the value of the `synlist` attribute above is the number of entries in the list, while the second argument is the actual list of entries. The rule for processing the SYN+ACK message at the clients is as follows.

```
<name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:false> (C(i)← SYN+ACK(SN,N,M)) T
 ⇒                <name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:true> [d+T,(SN←
ACK(C(i),M))] T  .
```

The rule is self-explanatory; the only significant point to be noted is that the attribute `connected` is set to true after processing the SYN+ACK message. Since the clients replicate their requests to account for random dropping of packets at the server, it is possible for them to receive a SYN+ACK message for a connection that has already been established. Such SYN+ACK messages are simply ignored as follows.

```
<name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:true> (C(i)← SYN+ACK(SN,N,M))
 ⇒ <name: C(i)|isn:N,repcnt:CNT,sendto:SN,connected:true> .
```

In contrast to the honest clients, the attacker ignores all the SYN+ACK messages that it receives from the server S.

```
<name: AN|sendto:SN > (AN← SYN+ACK(SN,N,M)) ⇒ <name: AN|sendto:SN >  .
```

Finally, the initial configuration of the system is

```
< name: AN | ... >   [ t1 , < name: C(1) | ... > ] [ t2 , < name: C(2) | ... > ] ...
    [tn , < name: C(N) | ... > ] < name: SN | ... >
```

where $t_1, \ldots, t_n$ are all distinct and positive. Note that, since all the clients are scheduled at different times, it follows [3] that the system does not contain any un-quantified non-determinism, which is essential for statistical analysis to be possible.

6 Analysis

We have successfully used the statistical model-checking tool VESTA [13, 14] to verify various desired properties of the probabilistic model in Section 5. In the following, we first describe the tool VESTA and its integration with PMAUDE. We then elaborate on the verification of one important property of the 3-way handshake protocol presented in the previous section.

The integration of PMAUDE and VESTA is described in detail in [3]. In the integrated tool, we assume that VESTA is provided with a set of sample execution paths generated through the discrete-event simulation of a PMAUDE specification with no non-determinism. We assume that an execution path that appears in our sample is a sequence $\pi = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \cdots$, where s_0 is the unique initial state of the system, s_i is the state of the system after the i^{th} computation step (rewrite), and t_i is the difference of global time between the states s_{i+1} and s_i. We also assume that there is a labelling function L that assigns to each state s_i a set of atomic propositions that hold in that state; the set of atomic propositions are all those that appear in the property of interest (see below). Thus, $L : S \rightarrow 2^{AP}$, where AP is a set of relevant atomic propositions and S is the set of system states. In PMAUDE, this labelling function is defined as an operator that maps terms representing states to sets of atomic propositions.

In VESTA, we assume that the properties are expressed in a sublogic of Continuous Stochastic Logic – CSL (without stationary state operators). CSL was introduced in [1] as a logic to express probabilistic properties. The syntax and the semantics of the logic and the statistical model-checking algorithm for CSL are described in [13, 14]. In our experiments, we model checked the following property expressed in CSL for different values of the attacker rate r_A.

$$\mathbf{P}_{\leq 0.01}(\Diamond(\textit{successful\_attack}()))$$

where *successful_attack*() is true in a state if the SYN cache of S is full, i.e., the attacker has succeeded in launching the SYN flood attack. The property states that the probability that eventually the attacker A successfully fills up the SYN cache of S is less than 0.01.

The results of model-checking are shown in the following table for two cases: in the absence of DoS counter-measure and in the presence of DoS counter-measure with the parameter p set to 0.9. In all experiments, we used scaled down parameters so that our experiments could be completed in a reasonable amount of time. Specifically, we used a SYN cache size of 10,000, cache timeout of 10 seconds, and 100 clients. The experiments were carried out on 1.8 GHz Xeon Server with 2 GB RAM and running Mandrake Linux 9.2.

Model-checking $\mathbf{P}_{\leq 0.01}(\Diamond(\textit{successful\_attack}()))$		X's attack rate (SYNs per second)								
		1	5	64	100	200	400	800	1000	1200
$p = 0.0$ (No counter-measure)	result	F	F	F	T	T	T	T	T	T
	time (10^2 sec)	47	87	280	605	183	183	182	182	181
$p = 0.9$ (With counter-measure)	result	F	F	F	F	F	F	F	T	T
	time (10^2 sec)	68	75	217	328	896	3102	11727	2281	1781

The results show that in the presence of DoS counter-measure with $p = 0.9$, S can sustain an attack from A with attack rate 10 times larger than that in the case of no counter-measure. Therefore, the results validate our hypothesis that *selective verification* can be used as an effective counter-measure for DoS attacks.

To gain more insight into the probabilistic model, we realized that model-checking is not sufficient. Specifically, we found the *true* (T) and *false* (F) answers given by the model-checker is not sufficient to understand the various quantitative aspects of the probabilistic model. For example, we wanted to know the expected number of clients that get connected in the presence of SYN flood attack. Therefore, in addition to model-checking, we used a query language called *Quantitative Temporal Expressions* (or QuaTEx in short). The language is mainly motivated by probabilistic computation tree logic (PCTL) [8] and Eagle [4]. In QuaTEx, some example queries that can be encoded are as follows:

1. What is the expected number of clients that successfully connect to `S` out of 100 clients?

2. What is the probability that a client connected to `S` within 10 seconds after it initiated the connection request?

A detailed discussion of the QuaTEx is beyond the scope of this paper. However, we provide a brief introduction of QuaTEx in the Appendix.

We evaluated the following QuaTEx expression with different values of the attacker rate r_A.

$$\texttt{CountConnected}() = \underline{\text{if}}\ \textit{completed}()\ \underline{\text{then}}\ \textit{count}()\ \underline{\text{else}}\ \bigcirc\,(\texttt{CountConnected}())\ \underline{\text{fi}};$$
$$\underline{\text{eval}}\ \mathbf{E}[\texttt{CountConnected}()]$$

In this expression, $completed()$ is true in a state if all the clients C_i have either sent all of their SYN packets or have managed to connect with S. The expression $count()$ in a state returns the number of clients that have successfully connected to S. The expression queries the expected number of clients that eventually connect with S in the presence of DoS attack by the attacker A.

The results of evaluating the above expression for different values of attacker rate r_A are plotted in Figure 2. The results show that most of the clients get connected as long as the attacker does not manage to fill up the SYN cache buffer. However, as soon as the attacker's SYN rate becomes high enough to fill the SYN cache buffer, none of the clients gets connected. The plot also illustrates that with *selective verification* the server can withstand an order of magnitude higher SYN flood rates than without.

7 Conclusions

We have presented a general framework for verification of DoS properties of communication protocols. We are able to express and prove key properties, but performance limitations of the automated system in our current formulation require us to use scaled down version of parameters that arise in practice. Addressing these efficiency limitations and verifying the properties for general systems remain future work objectives.

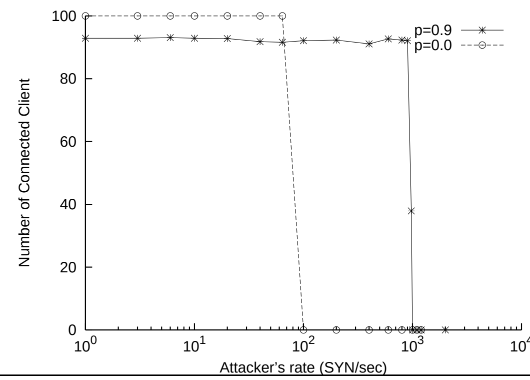

Figure 2: Expected number of clients out of 100 clients that get connected with the server under DoS attack

References

[1] A. Aziz, K. Sanwal, V. Singhal, and R. K. Brayton. Verifying continuous-time Markov chains. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102, pages 269–276.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] G. Agha, J. Meseguer, and K. Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05)*, 2005.

[4] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*, pages 44–57. Springer, January 2004.

[5] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. *Lecture Notes in Computer Science*, 1462, 1998.

[6] D. Dolev and A. C. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, 1983.

[7] C. A. Gunter, S. Khanna, K. Tan, and S. Venkatesh. Dos protection for reliably authenticated broadcast. In *Network and Distributed System Security (NDSS '04)*. Internet Society, 2004.

[8] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[9] N. Kumar, K. Sen, J. Meseguer, and G. Agha. Probabilistic rewrite theories: Unifying models, logics and tools. Technical Report UIUCDCS-R-2003-2347, Univ.of Illinois at Urbana-Champaign, 2003.

[10] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model for probabilistic distributed object systems. In *Proceedings of 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'03)*, volume 2884 of *LNCS*, pages 32–46, 2003.

[11] N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.

[12] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. MIT Press, 1993.

[13] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *16th conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 202–215, 2004.

[14] K. Sen, M. Viswanathan, and G. Agha. On statistical model checking of stochastic systems. In *17th Conference on Computer Aided Verification (CAV'05)*, LNCS (To Appear). Springer, 2005.

A QuaTEx

We introduce the notation that describes the syntax and the semantics of QuaTEx followed by a few motivating examples. Then we describe the language formally, along with an example query that we have used to investigate if the DoS free 3-way TCP/IP handshaking protocol model meets our requirements. The results of our query on various parameters are given in Section 6.

We assume that an execution path is an infinite sequence

$$\pi = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots$$

where s_0 is the unique initial state of the system, typically a term of sort `Config` representing the initial global state, s_i is the state of the system after the i^{th} computation step. If the k^{th} state of this sequence cannot be rewritten any further (i.e. is absorbing), then $s_i = s_k$ for all $i \geq k$.

We denote the i^{th} state in an execution path π by $\pi[i] = s_i$. Also, denote the suffix of a path π starting at the i^{th} state by $\pi^{(i)} = s_i \rightarrow s_{i+1} \rightarrow s_{i+2} \rightarrow \cdots$. We let $Path(s)$ be the set of execution paths starting at state s. Note that, because the samples are generated through discrete-events simulation of a PMaude model with no non-determinism, $Path(s)$ is a measurable set and has an associated probability measure. This is essential to compute the expected value of a path expression from a given state.

A.1 QuaTEx through Examples

The language QuaTEx, which is designed to query various quantitative aspects of a probabilistic model, allows us to write temporal query expressions like temporal formulas in a temporal logic. It supports a framework for parameterized recursive temporal operator definitions using a few primitive non-temporal operators and a temporal operator ($\bigcirc$). For example, suppose we want to know *"the probability that along a random path from a given state, the client $A(0)$ gets connected with B within 100 time units."* This can be written as the following query

$$\begin{array}{l} \texttt{IfConnectedInTime}(t) = \underline{\text{if}}\ t > \mathit{time}()\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ \underline{\text{if}}\ \mathit{connected}()\ \underline{\text{then}}\ 1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \underline{\text{else}}\ \bigcirc\ (\texttt{IfConnectedInTime}(t))\ \underline{\text{fi}}\ \underline{\text{fi}}; \\ \underline{\text{eval}}\ \mathbf{E}[\texttt{IfConnectedInTime}(\mathit{time}() + 100)]; \end{array}$$

The first four lines of the query define the operator $\texttt{IfConnectedInTime}(t)$, which returns 1, if along an execution path $A(0)$ gets connected to B within time t and returns 0 otherwise. The state function *time()* returns the global time associated with the state; the state function *connected*() returns true, if in the state, $A(0)$ gets connected with B and returns false otherwise. Then the state query at the fifth line returns the expected number of times $A(0)$ gets connected to B within 100 time units along a random path from a given state. This number lies in $[0, 1]$ since along a random path either $A(0)$ gets connected to B within 100 time units or $A(0)$ does not get connected to B within 100 time units. In fact, this expected value is equal to the probability that along a random path from the given state, the client $A(0)$ gets connected with B within 100 time units.

A further rich query that is interesting to our probabilistic model is as follows

$$\begin{array}{l} \texttt{NumConnectedInTime}(t, \mathit{count}) = \underline{\text{if}}\ t > \mathit{time}()\ \underline{\text{then}}\ \mathit{count} \\ \qquad\qquad \underline{\text{else}}\ \underline{\text{if}}\ \mathit{anyConnected}()\ \underline{\text{then}}\ \bigcirc\ (\texttt{NumConnectedInTime}(t, 1 + \mathit{count})) \\ \qquad\qquad\qquad\qquad\qquad \underline{\text{else}}\ \bigcirc\ (\texttt{NumConnectedInTime}(t, \mathit{count}))\ \underline{\text{fi}}\ \underline{\text{fi}}; \\ \underline{\text{eval}}\ \mathbf{E}[\texttt{NumConnectedInTime}(\mathit{time}() + 100, 0)] \end{array}$$

In this query, the state function $\mathit{anyConnected}()$ returns true if any client $A(i)$ gets connected to B in the state. We assume that in a given execution path, at any state, at most one client gets connected to B, which is true with our probabilistic model. We use a simpler variant of this query in our experiments.

A.2 Syntax of QuaTEx

The syntax of QuaTEx is given in Fig. 3. A query in QuaTEx consists of a set of definitions D followed by a query of the expected value of a path expression $PExp$. In QuaTEx, we distinguish between two kinds

Q	$::=$	D $\underline{\text{eval}}$ $\mathbf{E[}PExp\mathbf{]}$;	$SExp$	$::=$	$c \mid f \mid F(SExp_1, \ldots, SExp_k) \mid x_i$
D	$::=$	set of $Defn$	$PExp$	$::=$	$SExp \mid \bigcirc N(SExp_1, \ldots, SExp_n)$
$Defn$	$::=$	$N(x_1, \ldots, x_m) = PExp$;			$\mid \underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}$

Figure 3: Syntax of QUATEX

$$
\begin{aligned}
(s)[\![c]\!]_D &= c \\
(s)[\![f]\!]_D &= f(s) \\
(s)[\![F(SExp_1, \ldots, SExp_k)]\!]_D &= F((s)[\![SExp_1]\!]_D, \ldots, (s)[\![SExp_k]\!]_D) \\
(s)[\![\mathbf{E[}PExp\mathbf{]}]\!]_D &= \mathbf{E[}(\pi)[\![PExp]\!]_D \mid \pi \in Paths(s)\mathbf{]} \\
(\pi)[\![\underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}]\!]_D &= \text{ if } (\pi[0])[\![SExp]\!]_D = \text{true then } (\pi)[\![PExp_1]\!]_D \text{ else } (\pi)[\![PExp_2]\!]_D \\
(\pi)[\![\bigcirc N(SExp_1, \ldots, SExp_m)]\!]_D &= \\
& (\pi^{(1)})[\![B[x_1 \mapsto (\pi[0])[\![SExp_1]\!]_D, \ldots, x_m \mapsto (\pi[0])[\![SExp_m]\!]_D]]\!]_D \\
\text{where} \quad & N(x_1, \ldots, x_m) = B \in D
\end{aligned}
$$

Figure 4: Semantics of QUATEX

of expressions, namely, *state expressions* (denoted by $SExp$) and *path expressions* (denoted by $PExp$); a path expression is interpreted over an execution path and a state expression is interpreted over a state. A definition $Defn \in D$ consists of a definition of a *temporal operator*. A temporal operator definition consists of a name N and a set of formal parameters on the left-hand side, and a path expression on the right-hand side. The formal parameters denote the *freeze formal parameters*. When using a temporal operator in a path expression, the formal parameters are replaced by state expressions. A state expression can be a constant c, a function f that maps a state to a concrete value, a k-ary function mapping k state expressions to a state expression, or a formal parameter. A path expression can be a state expression, a next operator followed by an application of a temporal operator already defined in D, or a conditional expression $\underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}$. We assume that expressions are properly typed. Typically, these types would be `integer`, `real`, `boolean` etc. The condition $SExp$ in the expression $\underline{\text{if}}\ SExp\ \underline{\text{then}}\ PExp_1\ \underline{\text{else}}\ PExp_2\ \underline{\text{fi}}$ must have the type `boolean`. The temporal expression $PExp$ in the expression $\mathbf{E[}PExp\mathbf{]}$ must be of type `real`. We also assume that expressions of type `integer` can be coerced to the `real` type.

A.3 Semantics of QUATEX

Next, we give the semantics of a subset of query expressions that can be written in QUATEX. In this subclass, we put the restriction that the value of a path expression $PExp$ that appears in any expression $\mathbf{E[}PExp\mathbf{]}$ can be determined from a finite prefix of an execution path. We call such temporal expressions *bounded* path expressions. The semantics is given in Fig. 4. $(\pi)[\![PExp]\!]_D$ is the value of the path expression $PExp$ over the path π. Similarly, $(s)[\![SExp]\!]_D$ is the value of the state expression $SExp$ in the state s. Note that if the value of a bounded path expression can be computed from a finite prefix π_{fin} of an execution path π, then the evaluations of the path expression over all execution paths having the common prefix π_{fin} are the same. Since a finite prefix of a path defines a basic cylinder set (i.e. a set containing all paths having the common prefix) having an associated probability measure, we can compute the expected value of a bounded path expression over a random path from a given state. In our analysis tool, we estimate the expected value through simulation instead of calculating it exactly based on the underlying probability distributions of the model. The exact procedure can be found at `http://osl.cs.uiuc.edu/~ksen/vesta2/`.