Timing analysis of TCP servers for surviving denial-of-service attacks

V Krishna Nandivada  Jens Palsberg
UCLA

Abstract—Denial-of-service attacks are becoming more frequent and sophisticated. Researchers have proposed a variety of defenses, including better system configurations, infrastructures, protocols, firewalls, and monitoring tools. Can we validate a server implementation in a systematic manner? In this paper we focus on a particular attack, SYN flooding, where an attacker sends many TCP-connection requests to a victim’s machine. We study the issue of whether a TCP server can keep up with the packets from an attacker, or whether the server will exhaust its buffer space. We present a tool for statically validating a TCP server’s ability to survive SYN flooding attacks. Our tool automatically transforms a TCP-server implementation into a timed automaton, and it transforms an attacker model, given by the output of a packet generator, into another timed automaton. Together the two timed automata form a system for which the model checker UPPAAL can decide whether a bad state, in which the buffer overruns, can be reached. Our tool has two advantages over simply testing the server implementation with a packet generator. First, our tool is an order of magnitude faster because of aggressive abstraction of the server code. Second, our tool can be applied to a variety of server software without having to install each one in the kernel of an operating system. Thus, a programmer of defensive measures against SYN flooding attacks can get rapid feedback during development.

I. INTRODUCTION

A. Background

Attacks on internet sites are becoming frequent. Increasingly sophisticated attacks on the websites of the SCO [37], the RIAA [18], the Al Jajeera [36], the CERT [28], and the White House [29] show that no site can hope to avoid denial of service (DoS) attacks. For health-care monitoring and diagnosis over the Internet, Nisley wrote that “a distributed Denial-of-Service attack on the monitoring center may prove fatal” [32]. In response, researchers and system administrators have built various degrees of defenses against DoS attacks, including systems with more resources, more restrictive protocols, firewalls, monitoring systems, and reactive systems [3], [43], [27], [15], [14], [38], [39], [35]. We can divide the defense measures into two categories: detection of denial of service attacks [43], [27], [39], [41], [8], and response to such attacks, either by trying to traceback the source [14], [23] or by managing the attack such that the impact can be reduced [30], [35], [45]. Much of the battle between attackers and defenders takes place at the TCP-level: more than 90% of the DoS attacks use TCP [31]. This paper is concerned with tools and methods to validate TCP-server implementations and answer questions such as:

Challenge: Can we determine efficiently whether a TCP server will survive a denial-of-service attack?

Our main contribution is to show that for a class of attacks such challenges can be answered efficiently and fairly accurately using timing analysis of the server code and a novel model of TCP.

B. The Problem

In this paper we focus on SYN flooding attacks which exploit that TCP [1] requires a three-way handshake to take place before data can be transmitted between a client and a server. In the first step of a three-way handshake, see Figure 1(a), the client sends a packet SYN_x to the server. The packet SYN_x is a TCP-connection request and x is a sequence number. Second, the server stores a representation of SYN_x in a buffer and responds to the client with two packets SYN_y and ACK_x+1. Third, for the connection to be established, the server needs to wait for an appropriate acknowledgment from the client, in the form of an ACK_y+1 packet. When the ACK_y+1 packet has been received, the server clears the buffer entry for SYN_x. If the server does not get a response from the client before a specified timeout time, then the server will time out the packet and clear the buffer entry.

Notice that when the server receives a SYN packet, the server will allocate a buffer entry. SYN flooding is
an attempt to exhaust the available buffer space such that SYN packets from valid clients will have to be rejected, thereby leading to denial of service to the valid clients. A malicious client can do SYN flooding by repeatedly sending SYN packets, see Figure 1(b), without ever sending ACK packets to complete the three-way handshake. The SYN packets will get stored in the buffer and unless packets time out fast enough, they will accumulate in the buffer and exhaust the available space. If the buffer space is unbounded, the situation is even worse: SYN flooding can lead to taking up all the space resources of the server.

SYN flooding attacks are easy to build and have a strong effect in terms of blocking the service to other clients. Not only web servers but also any system connected to Internet providing TCP-based services such as FTP or Mail servers are susceptible to SYN flooding attacks. The core problem is about timing: “can the TCP server keep up with the packets from the attacker, or will the TCP server exhaust its buffer space?”

C. Simulation-based Experiment

![Simulation-based experiment](image)

We begin with an experiment that examines a state-of-the-art simulation-based approach to validating TCP server implementations. The approach uses one of the many network simulators and integrated experimental environments [12], [22], [24], [44] to deploy and test the TCP-server implementation. Further, the approach simulates network traffic (from attackers and well-behaved clients) with a packet generator, and then it checks whether the server succumbs to attacks. We chose the TCP-server implementation lwIP [19], the accompanying experimental environment simhost, the packet generator D-ITG, [9] and the packet sender Spak [40]. In the experiment, we set the buffer size to five, we used packet sequences of ten packets, and we tried 10,000 combinations of:

1) the average inter-packet arrival time, that is, the mean of a normal distribution with the standard deviation being 25% of the average, and
2) the timeout time, that is, the length of time the server will keep a SYN packet in its buffer before the server removes the packet.

Furthermore we repeated the experiment ten times. Figure 2 presents the results: red means that the server succumbed to the attack at least once, while green means that the server did not succumb to the attack. (On black-and-white hardcopies, green is represented by small triangles, and red by small dots.) On a system that has dual Intel Xeon CPUs running at 3.06GHz with 512 KB of cache and 4GB of main memory, the ten runs took a total of around 850 minutes. We limited ourselves to packet traces of ten packets because for larger and more realistic traces (of sizes say greater than 100) we could not run even one instance of the simulator to completion, even after running it for more than 8 hours.

The simulation-based approach produces a graph that is useful for a system administrator who wants to tune the timeout time and buffer size and for a developer of TCP servers who wants to gauge the quality of an implementation. Additionally, recent QOS-regulation approaches enable a system to dynamically tune the packet inter-arrival time by filtering the traffic to a TCP server [21]. However, the simulation-based approach is time consuming, both because of the simulation time itself and because the approach requires the OS kernel to be recompiled and redeployed in the experimental environment. Further, the presence of monitoring code in the system impacts the temporal behavior, reducing the accuracy of the results. Can we do better using timing analysis?

D. Our Results

![Block diagram for ATASYN](image)

We present a tool called ATASYN (Analyzer for Timed Applications—SYN flood detection) for determining whether a TCP server will survive a SYN flooding attack. Our tool takes the same input as the simulation-based experiment reported above, namely a
TCP-server implementation and a packet sequence. Our tool is based on timing analysis and can efficiently derive good approximations of graphs such as the one in Figure 2. ATASYN embodies three ideas:

\[
\text{ATASYN} = \text{model of TCP as a timed automaton} \\
+ \text{timing analysis} \\
+ \text{real-time model checking.}
\]

As a result, instead of running a slow and highly accurate simulation using simhost, we can now run a fast and fairly accurate simulation using ATASYN.

Figure 3 shows a block diagram of ATASYN’s two phases. The analyzer phase uses timing analysis to automatically abstract the code of a TCP-server implementation into a timed automaton [6], and it transforms straightforwardly a packet sequence into another timed automaton. The validator phase combines the two timed automata into a system for which the real-time model checker UPPAAAL [11] can decide whether a bad state, in which the buffer overruns, can be reached. ATASYN is an order of magnitude faster than the simulation-based approach because of aggressive abstraction of the TCP-server code. Moreover, ATASYN can be applied to a variety of server software without having to install each one in the kernel of an operating system. Thus, a programmer of defensive measures against SYN flooding attacks can get rapid feedback during development. ATASYN follows in the footsteps of much work on temporal properties [7], [5], [26], [10].

When comparing the results of simulation and ATASYN, we will use the following terminology:

- **False positive**: The simulation does not report a successful attack, but ATASYN does.
- **False negative**: The simulation does report a successful attack, but ATASYN does not.

Ideally, there would be no false positives and no false negatives. ATASYN reports 2% false positives and 9% false negatives in our experiments. The high speed and low error rate of ATASYN makes it a practical tool for the working system administrator.

The main technical challenge is to devise an abstraction of TCP which leads to an efficient and accurate SYN-flood vulnerability detector, while abstracting away computational details that are irrelevant to buffer overflow. Our new timed automaton for modeling TCP has a number of states which is linear in the size of the buffer. We use worst-case execution time (WCET) analysis [33], [34], [2], [16], [13], [25], [20] to compute the timings needed in the timed automaton.

In the following section, we describe how we abstract TCP code into a timed automaton. In section III we present a timed automaton for handling multiple packets, and in section IV we show how to represent an attacker as a timed automaton. Finally, in section V we present our experimental results.

II. From TCP Code to a Timed Automaton

Most TCP servers have three main components: a packet interrupt handler, packet processing routines, and a timer interrupt handler. When a TCP server receives a packet, the server will run a packet interrupt handler, which in turn call the packet processing routines. With regular intervals, the timer interrupt handler removes timed-out packets from the buffer. The timer interrupt handler also has the task of firing a packet-handler interrupt when it identifies a newly arrived packet in the OS-queue. Note that the OS-queue is a data structure different from the SYN-buffer.

ATASYN maps a TCP implementation to a timed automaton of the form shown in Figure 4. The automaton in Figure 4 models how to handle a single packet. In the following section we will show how to extend the automaton to handle multiple packets. Notice that Figure 4 shows an informal version of the timed automaton on the left and the actual timed automaton on the right. Basic information about timed automata can be found in the appendix.

The automaton has seven states. Notice that six of the states have two labels, while the seventh state is called *Timeout*. For the states with two labels, the first label is a name of the state (the label is one of \( A_1, A_2, A_3, A_4, A'_4, A_5 \)), while the second label denotes a WCET (the label is one of \( C_1, C_2, C_3, C_4, T_2 \)). \( A_1 \) models the packet interrupt handler; we use \( C_1 \) to denote the WCET of \( A_1 \). \( A_2 \) models the packet processing routines; we use \( C_2 \) to denote the WCET of executing code from the beginning of \( A_1 \) to the end of \( A_2 \). \( A_3 \) models the clearing of the OS-queue entry; we use \( C_3 \) to denote the WCET of executing code from the beginning of \( A_1 \) to the end of \( A_3 \). \( A_4 \) and \( A'_4 \) both model a run of the timer interrupt handler; we use \( C_4 \) to denote the WCET of executing the interrupt handler. The hardware timer sends interrupts at a regular interval of length \( T_p \) and this interrupt wakes up the timer interrupt handler. \( T_p \) is a constant that typically is embedded in the TCP-implementation and must be identified by the user of ATASYN. All of \( C_1, C_2, C_3, C_4 \) includes the time to execute the timer interrupt handler, possibly several times. \( A_5 \) models the waiting for an ACK packet; the maximum wait time is the timeout time for SYN packets, which we denote by \( T_o \). The last state, labeled *Timeout*, is reached when at least one packet times out.

Let us now consider how to obtain the WCETs \( C_1, C_2, C_3, C_4 \) from the TCP-server implementation lwIP, which is written in C. ATASYN relies on that the user identifies the beginning and end of each of
the four corresponding pieces of code in the C source code. The user does the identification by wrapping each code piece in a function. It is difficult to obtain WCET information directly from a high level language like C. In contrast, we can get good timing estimates for the object code generated by gcc or any other C compiler. However, because of the code motion which is part of gcc’s optimization phase, it can be difficult to identify in the object code where a particular functionality is implemented. So, we take a middle approach: we first use gcc to compile the C-code to register transfer language (RTL), which is a format used internally in the gcc compiler. It is the backend of gcc that translates the front end’s abstract syntax tree into RTL. At the RTL level, it is straightforward to do WCET analysis. Since there is a direct mapping from RTL instructions to machine code the timing estimates can be done with more confidence than for C. We have added a phase in gcc just before the final phase that emits assembly code. While the actions of the assembler and linker may affect the timing analysis, our results show that for our application, the overall effect is small.

We employ a well-known WCET analysis [20] which takes the pipelining architecture of the Xeon processor into account. Our implementation of the WCET analysis is conservative in several ways, including that it does not take into account that Xeon is a super-scalar architecture, or the effect of cache or data dependencies. We chose that particular WCET analysis because it leads to good overall results. It remains to be seen whether a better WCET analysis can significantly improve our results.

In the TCP/IP implementation lwIP, each loop in the packet handler and the timer interrupt handler iterates over the buffer. So as a simple static estimate of the number of loop iterations, we use the buffer size.

When a packet is received by the TCP server, the packet interrupt handler (labeled $A_1$) is invoked. In Figure 4 we want to start the clock from that point so we reset the clock variable $T$ to zero. In timed automata, transition from one state to another requires that the guard on the edge be satisfied; accordingly we set the guards in our transitions such that it semantically agrees to the informal automaton on the left. At the end of state $A_1$, the clock is set to a value greater than sum of WCET of the packet interrupt handler, and the time to execute the invoked timer interrupt handlers. Similarly, at the end of state $A_2$, the clock is set to value greater than the time required to execute the code corresponding to these states and the time to run the timer interrupt handler every time it was invoked during the execution.

The modeling of the timer interrupt handler requires further explanation. The transitions from $A_1, A_2, A_3, A_5$ to Timeout all model the case where the timer interrupt handler runs and times out at least one packet. The transitions from $A_2$ to $A_4$ and from $A_5$ to $A'_4$ model the case where the timer interrupt handler runs but does not time out any packets. It would be semantically correct to add such transitions to $A_1, A_3$, but we left them out as an optimization. The reason why the optimization is correct is that at $A_1, A_3$, the packet interrupt handler is
disabled, so there it would have no immediate effect to fire a packet-handler interrupt. Furthermore, if a packet times out while executing $A_1$, it will be timed out when execution reaches $A_2$; similarly for $A_3$. Note that while executing the timer interrupt handler, none of the interrupts are enabled.

In summary, the user of ATASYN must identify four pieces of code, the timer period, and the buffer size, and ATASYN then maps the packet handler to a timed automaton. Our model is conservative because 1) the WCET analysis provides upper bounds on the execution time and 2) our model ensures that the timer interrupt handler is called as least as many times as the timer interrupt handler is invoked in the actual server. Notice that the number of nodes in the timed automaton is fixed and independent of the number of lines of code in the TCP implementation.

III. Handling Multiple Packets

ATASYN abstracts a server that has multiple buffer entries and handles multiple packets into a timed automaton. In this section we extend the automaton given in section II to complete the abstraction. Note that our execution model consists of a single processor with an OS capable of running multiple threads. Hence at any one particular time, only one thread can be active.

![Fig. 5. Complete timed automaton for a server.](image)

Let us now look at the server automaton given in Figure 5. We need to be able to analyze and reason about several copies of the code in packet processing routines at the same time, namely, as many as there are buffer entries. For each buffer entry, we replicate the model shown in previous section. In addition, there are edges connecting each replica and a self loop on the state $A_1$ in each replica. Each such replica is called a zone, and $zone_i$ and $zone_{i+1}$ are connected, signifying that when $i^{th}$ packet is being processed, the next packet $(i + 1^{th})$ can come and the server may process this new packet. Each $zone_i$ has an individual timer $T_i$ keeping track of the time in that zone. Along with these, we also have a global timer $GTimer$ that keeps track of global time. $GTimer$ is reset to zero in the first state of the $zone_1$.

The server, on getting the first SYN packet, goes to the state $A_2$. Now either in this state or in $A_5$, if the server finds another SYN packet for a fresh connection, then the server starts working on the second packet. The guard $Grd$ on that edge is given by:

$$\text{InputIsReady} \land GTimer \leq T_o.$$  

We implement the macro InputIsReady by a set of synchronization constructs between the server automaton and the attacker automaton (we omit the details).

The same strategy continues while processing the rest of the packets. Finally, if the server has already filled up $m$ entries, where $m$ is the size of the buffer, and another SYN packet arrives for a new connection, then the packet has to be dropped and we have an effective SYN flooding attack. In the automaton we model that by reaching the final state SYNFlood. If we reach SYNFlood, then the attacker can continue the attack and keep all the servers’ buffer entries occupied, resulting in a successful denial-of-service attack.

The server has many states from which it cannot be SYN flooded or which are not important when studying the SYN flooding attack. We abstract all those states into one dummy node $Timeout$. For example, from the states $A_3$, and $A_5$ in the first zone, and from $A_1$, $A_2$, $A_3$, and $A_5$ in the other zones we have an edge to $Timeout$ with the following guard that transfers control to the $Timeout$ state, if the first packet times out:

$$GTimer \geq T_o.$$  

We abstract the possible dropping of a packet by setting up a self loop in the state $A_1$ with a guard $Le$ given by:

$$\text{InputIsReady} \land GTimer \leq T_o.$$  

We also have some more constraints in the automata to model the OS Queue and some synchronization constructs for the abstraction of a loss free network (we omit the details).

The actual server inserts and removes packets from the buffer. The server removes a packet if the packet times out or if the server receives a reset packet (RST) or an acknowledgment packet (ACK). If we want to model all possible states of the buffer, then we would need exponentially many states in the server automaton, one for each subset of the packets. Fortunately, for the purposes of ATASYN, we don’t need exponentially many states. The reason is that we distinguish between packet sequences solely on the basis of whether they can reach the SYNFlood state or not. The question that we ask of UPPAAL for a given packet sequence is: “does there exist a run of the automaton which reaches the
SYNFlood state?" A packet sequence that cannot reach the SYNFlood state will count as an unsuccessful attack. Let us consider a packet sequence containing a packet that in the actual server gets inserted into the buffer and later times out and gets removed. Suppose also that the packet sequence leads to SYNF flooding. In our server automaton, at least two paths are possible. One path will process the packet in the appropriate zone and later, when that packet times out, go to the Timeout state. So, that path does not demonstrate the SYNF flooding. However, there is another path in which we immediately drop the packet, using a self loop, and now there is no reason for going to the Timeout state, so instead the server automaton will process the other packets and end up in the SYNFlood state. When the server automaton drops a packet using a self loop in one zone, it continues in the same zone when the next SYN packet comes. Thus, the server automaton contains only forward transitions, i.e., transitions from zone $i$ to $i + 1$.

For example, let us consider a packet sequence $p_1, p_2, \ldots, p_{m+1}$ which leads to SYNF flooding. Suppose that while control is in zone $i$, processing packet $p_1$, the first packet $p_1$, stored in the buffer in zone $1$, times out. In that case, we use a transition to the Timeout state. However, the packet sequence $p_2, \ldots, p_{m+1}$ can still lead to SYNF flooding. We can reach the SYNFlood state by the following actions: packet $p_1$ is dropped in zone $1$, then zone $1$ gets packet $p_2$, zone $2$ gets packet $p_3$, and eventually we reach the SYNFlood state.

In summary, we avoid an automaton of exponential size by (1) asking whether the SYNFlood state is reachable, (2) using self loops to drop packets, and (3) using transitions to the Timeout state. The size of our server automaton is linear in the buffer size. For a buffer of size $m$, the server automaton has $6 \times m + 2$ states.

IV. FROM ANAttacker TO A Timed Automaton

We assume that the attacker (or, in general, the network environment around the server) sends packets with a delay controlled by a statistical distribution. The distribution is “tester chosen”, can be varied, and does not influence how we model an attacker. Given a distribution we use a packet generator to compute the packet inter-arrival time for a pre-determined number of packets.

The generated packet sequence is the one seen by the server. For the server, all the packets lost during transmission are not of any concern. So, the attacker starts by sending a SYN packet, and then it sends more SYN packets at the pre-decided intervals. The attacker sends the next packet only after the server has read the last packet because all the server cares about are the packets that comes to its notice. The attacker stops after sending the pre-decided number of packets. Notice that ATASYN’s interpretation of a packet sequence is different from what happens when we send the same packet sequence to simhost. The reason is that the network may loose packets, deliver them in a different order, and change the intervals. This difference will eventually be part of the reason why simhost and ATASYN give slightly different results.

We use the distributed internet traffic generator D-ITG [9] as the basis for generating SYN packets at random intervals. D-ITG can generate packets with delays based on different statistical distributions. Our aim is to feed some large number of SYN packets, generated with delays based on some statistical distribution, to the server and see the effect of the attack. So, we use the D-ITG to generate packet inter-arrival times and compute the time for sending the $i^{th}$ packet ($PAT(i)$). Since we want to fire the next packet only after the previous packet is read and the server is ready for the next packet, the packet generator process (represented as an attacker timed automaton) waits for communication from the server timed automaton to release the next packet.

To generate a timed automaton for the attacker, we generate one node for each packet and connect the nodes for packets $i$ and $i + 1$ by an edge. We also generate a special node start node for the attacker and connect it to the node corresponding to the first packet. We set $PAT(start)$ to be zero. For each node $i$ in the attacker timed automata, we set an invariant $GTimer \leq PAT(i)$. Each edge between two neighbors $i$ and $j$, has a guard, $GTimer \geq PAT(i)$, a synchronizing guard $inputIsRead?$, and an assignment to the global variable $input = SYN$.

The model for the attacker is shown in Figure 6. The automaton will stop after sending a pre-determined number ($n$) of packets. The number of nodes in the attacker automaton is $n + 1$, and the number of edges is $n$.

After the timed automata for the server and attacker
have been generated, and given values for the packet inter-arrival time and the timeout time, ATASYN runs UPPAAL. Running UPPAAL will check if the server automaton will go to the SYN Flood state, which would indicate SYN flooding.

V. EXPERIMENTAL RESULTS

We have implemented our analysis in the optimization phase of gcc 3.3 [4] for the x86 architecture, before gcc reorders the basic blocks.

![Figure 7](output_by_ATASYN.png)

**Fig. 7.** Output by ATASYN.

We have used ATASYN to test the light-weight TCP/IP protocol implementation lwIP [19]. The lwIP implementation can run with or without the underlying OS. The focus of the lwIP implementation is to reduce the RAM usage while still having a full scale TCP, making lwIP suitable for use in embedded systems. The complete lwIP TCP/IP implementation is around 18K lines with around 300 functions. Of this the TCP SYN-packet-processing-related code is around 2500 lines of code with around 30 functions. For the purposes of this paper we set the buffer size to 5. We have tested ATASYN with buffer sizes up to 100 and it scales well.

![Figure 8](output_by_ATASYN_with_false_positives.png)

**Fig. 8.** Output by ATASYN, with false positives.

We have run ATASYN on the same computer, the same 10,000 combinations of inter-packet arrival time and timeout time, and the same packet sequences for each of the 10,000 points that we used for the experiment with simhost reported in Section I. Figure 7 presents the results: red means that the attack is successful, while green means that the attack is not successful. (On black-and-white hardcopies, green is represented by small triangles, and red by small dots.) We can compare Figure 2 and Figure 7 directly: ideally the figures would be identical or close to identical. Let us first examine Figure 7. In the red area, the attacks are successful. For a given packet inter-arrival time, it is better not to choose a timeout time such that we get a red point.

In the green area, the attacks are not successful. The region to the right of the red area agrees with the intuition that higher timeout time is affordable for environments with longer packet inter-arrival delays. Common sense says that if an attacker can send packets at a high rate, i.e., the packet inter-arrival time is small, then the timeout time should be set to a small value. However, notice the green region below timeout = 3.5 × 10^8. It is perhaps surprising to see that even for low packet inter-arrival delays, the server did not get SYN flooded. The main cause for that is the lower timeout time. Because the timeout time was so low, earlier packets are getting timed out before all the buffer entries could be filled. Still, the timeout time was large enough that the TCP server would accept some connections. It looks rather attractive to set the timeout time to about 3.5 × 10^8. It may be noted that even though this would not lead to SYN flooding, it may disallow connections to some valid clients whose packets do not come fast enough.

Note that the boundary between red and green is not sharp. This is because for each average packet inter-arrival time we get sequences of delays based on a statistical distribution. A constant delay packet generator would give a clearer demarcation between the regions.

ATASYN takes around 4 minutes to generate the graph shown in Figure 7. In contrast, each of the ten runs of simhost to generate the graph shown in Figure 2 took around 85 minutes. So, ATASYN is more than 20 times faster than simhost. ATASYN spent less than one second to produce the timed automaton from the lwIP implementation, and most of the 4 minutes on running UPPAAL. For packet sequences of length 100, simhost did not terminate after 8 hours. In contrast, for packet sequences of length 500, ATASYN generates a graph like the one in Figure 7 in around 40 minutes.

Let us now compare the accuracy of ATASYN relative to simhost. In Figure 8, green means that simhost and ATASYN agree on green, while red denotes a false positive.
positive. A false positive is a case for which simhost not report a successful attack, but ATASYN does. We
found 230 false positives in the sample space of 10,000
points, that is, 2.3%. In other words, if ATASYN reports
a successful attack, then there is nearly a 98% chance
that simhost will agree. Notice that the false positives are
distributed in a seemingly random fashion. One of the
main reasons for the false positives is that when a packet
sequence is sent to simhost, the network may deliver the
packets to TCP layer of simhost with different intervals
between the packets. In contrast, the packet sequence
sent to ATASYN represent the intervals between when
packets are made available to the TCP layer.

In Figure 9, green means that simhost and ATASYN
agree on green, red means that simhost and ATASYN
agree on red, while a blue circle denotes a false negative.
A false negative is a case for which ATASYN does not
report a successful attack, but simhost does. We found
916 false negatives in the sample space of 10,000 points,
that is, 9.16%. In other words, if ATASYN reports does
not report a successful attack, then there is nearly a 91%
chance that simhost will agree. Notice the blue points
along the y-axis; they represent a curious anomaly
that fortunately only occurs when the packet inter-arrival
times are close to 0.

The 2.3% false positives indicate that ATASYN is
particularly effective at determining when a TCP server
is vulnerable to SYN flooding. The 9.16% false negatives
serve as a reminder that ATASYN cannot entirely replace
simhost. ATASYN is most valuable in cases where a
developer or an administrator of a TCP server wants to
quickly gauge the degree of vulnerability.

VI. SUMMARY

The main contributions and features of ATASYN are:

- **Fast:** ATASYN runs an order of magnitude faster
  than doing similar validation by running the actual
  server.

- **Accurate** The suggestions/warnings given by ATAS-
  YN are fairly accurate. The graph produced by
  ATASYN is highly comparable to that produced by
  running the actual server.

- **Modular:** There is no need to have or modify the
  entire OS kernel: the TCP server is analyzed in
  isolation.

- **Scalable:** ATASYN can test the server against
  large sequences of input packets within reasonable
  amount of time.

- **Graphical:** ATASYN produces a graph that shows
  whether a server can survive SYN flood attacks.

- **Uses off-the-shelf tools:** ATASYN uses gcc, UP-
  PAAL, and gnuplot.

- **Needs few user annotations:** A few, simple-to-give
  user annotations are needed.

We believe that our approach is promising and may
be useful in the context of other denial-of-service at-
tacks. Our technique is fairly independent of the high-
level language in which the TCP server is implemented
because the timing analysis and the abstraction into a
timed automaton are done on RTL code, which is an
intermediate format close to assembly language. Our
technique is also fairly independent of the choice of the
real-time model checker. Model checkers that could take
the place of UPPAAL include KRONOS [17] and RED
[42]. It remains to be seen whether the use of a different
model checker can increase the speed of ATASYN.

Our tool ATASYN is available from our website at
http://compilers.cs.ucla.edu/atasyn.

Acknowledgments. We thank Christopher Telfer for
help with various networking tools. We thank Kevin
Chang and Ben Titzer for helpful comments on a draft on
the paper. We were supported by two National Science
Foundation ITR Awards number 0112628 and 0427202.

APPENDIX: TIMED AUTOMATA

Real-time systems can be modeled by timed automata.
A timed automaton is a finite state automaton with
integer-valued clocks. The states are represented by \((l, u)\)
where \(l\) is a control node and \(u\) is a clock assignment,
i.e., the current values of all the clocks. All the clocks of
a system start at the same instant from 0 and then they
proceed at the same rate. Their values can be tested (by
comparing them to natural numbers), they can be reset,
and they can be assigned a natural number. Guarantees
about timing are enforced by clock constraints (which
are guards on transitions and invariants on nodes). A
clock constraint in UPPAAL is a constraint consisting
of zero or more of the following three components: a
guard that needs to be true for the transition to take
place, a channel read/write synchronization command, and an assignment statement that assigns to a timer or a variable. If the transition has a channel read command, then for the transition to succeed some process must have written to the channel and that data is available to be read. Once the data is read, the next read will fail until some process has written again.

REFERENCES