# Game-Based Analysis of Denial-of-Service Prevention Protocols

Ajay Mahimkar
Department of Electrical & Computer Engineering
The University of Texas at Austin
Austin, TX 78712 U.S.A.
mahimkar@cs.utexas.edu

Vitaly Shmatikov
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 U.S.A.
shmat@cs.utexas.edu

## Abstract

*Availability is a critical issue in modern distributed systems. While many techniques and protocols for preventing denial of service (DoS) attacks have been proposed and deployed in recent years, formal methods for analyzing and proving them correct have not kept up with the state of the art in DoS prevention. This paper proposes a new protocol for preventing malicious bandwidth consumption, and demonstrates how game-based formal methods can be successfully used to verify availability-related security properties of network protocols.*

*We describe two classes of DoS attacks aimed at bandwidth consumption and resource exhaustion, respectively. We then propose our own protocol, based on a variant of client puzzles, to defend against bandwidth consumption, and use the JFKr key exchange protocol as an example of a protocol that defends against resource exhaustion attacks. We specify both protocols as alternating transition systems (ATS), state their security properties in alternating-time temporal logic (ATL) and verify them using MOCHA, a model checker that has been previously used to analyze fair exchange protocols.*

## 1. Introduction

A fundamental design principle of the IP architecture is to keep the functionality inside the core network simple, pushing difficulty to the network endpoints. This principle, commonly referred to as the "end-to-end principle," has guided most of modern network protocol design. One of the undesirable side effects of this approach, however, has been the problem of malicious traffic aimed at denial of service (DoS). DoS traffic is caused by deliberate distributed flooding [32], worms such as Nimda, Code Red, and SQL Slammer [31, 40], port scanners, and spammers. The need to protect against or mitigate this flood of undesirable traffic has been recognized by both commercial and research groups. Since there are few disincentives for clients who contribute to this traffic, the standard mechanism has been to first detect the clients exhibiting suspicious behavior, and then install filtering rules on routers to block their traffic. While a variety of anti-DoS mechanisms and protocols have been proposed, relatively few of them have been analyzed using formal verification techniques that have been successfully applied to the analysis of other security properties such as secrecy and authentication.

Methods that filter traffic [6, 18, 19, 30, 28, 34, 35, 46, 37, 38, 39, 42, 47] look for known attack patterns or statistical anomalies in traffic patterns. They can be defeated by changing the attack pattern and masking the anomalies. Also, statistical approaches might filter out legitimate traffic, too. There is a need for preventive mechanisms that allow all legitimate traffic to reach the destination.

*Client puzzles* [12, 13, 14, 20, 41, 43, 44] are a promising technique that aims to provide service guarantees to legitimate clients. Clients get access to a service only after they prove their legitimacy. For each service request, the client is forced to solve a cryptographic "puzzle" before the server commits its resources. This imposes a large computational task on adversaries generating traffic in large quantities. The main idea behind client puzzles is that any client requesting service must allocate some of its own resources (processing time or memory) before the server commits its resources for the connection. This defends against attacks staged by a large number of zombie computers using authentic IP addresses, since existing DDoS tools are carefully designed not to disrupt the zombie computers so as to avoid alerting the machine owners to their presence.

In this paper, we discuss protocols that prevent two types of attacks: those aimed at *resource exhaustion* and *bandwidth consumption*, respectively. In a *resource exhaustion* attack, the attacker sends spurious requests to the server, causing the latter to devote all of its resources to processing the malicious requests and to start dropping requests from legitimate clients. An example of such an attack is the TCP SYN flooding attack (see, *e.g.*, [42]). An even more se-

vere attack is the *bandwidth consumption* attack in which the attacker floods the connection to the server, causing the latter to stop accepting, let alone processing requests from legitimate clients. Here the goal of the attack is to consume all available network bandwidth by flooding it with a huge number of properly or improperly formed requests.

We propose a new system to defend the network against bandwidth consumption attacks, and use the JFKr key exchange protocol [3] as an example of a protocol that defends against resource exhaustion attacks. We then use game-based analysis techniques to formally state the availability guarantees provided by the protocols, and verify them using the MOCHA model checker.

**Contributions and overview.** In this paper, we formally describe two existing anti-DoS protocols [13, 43] which are based on client puzzles. We will refer to these protocols as *basic* client-puzzle protocols because they prevent resource exhaustion, but not bandwidth consumption attacks. We then propose our own protocol to defend against bandwidth consumption attacks. Our protocol operates at the IP layer, and provides an efficient defense since it stops all malicious traffic in the *intermediate* network, before it can reach its final destinations.

The protocols are specified and verified using a game-based formalism and the associated model checker called MOCHA [5]. The interaction between the attacker and the defending server or filtering node is modeled as a two-player strategic game. In this game, the server's strategy is characterized by the difficulty level of the puzzle it generates and presents to the client, whereas the DoS attacker's strategy is characterized by the amount of effort it can invest in solving the puzzles.

While there has been some previous work on formal modeling of anti-DoS protocols [25, 27, 29, 36, 45], we believe this work is the first to use a game-based framework, which is a natural fit for *adversarial* protocols such as those arising in DoS prevention, and also the first to focus on bandwidth- rather than resource-consumption attacks.

We believe that our distributed approach to DoS prevention is interesting from the viewpoint of protocol design and not just that of formal modeling. First, it does not require the formulation of attack signatures by which routers detect and/or filter suspicious traffic. Since ours is a preventive approach, it does not result in dropping any legitimate traffic. This is due to our use of client puzzles. The ability to solve puzzles separates legitimate clients from automatic attack tools that typically operate on zombie computers and are carefully designed to avoid performing computations that may alert machine owners to their presence. Second, our protocol is designed to work within the IP layer, in the intermediate network, preventing malicious traffic from reaching end servers and thus preventing bandwidth consumption attacks. Third, our protocol can be deployed incrementally.

The main contributions of the paper are as follows:

1. A novel distributed client puzzle protocol that prevents bandwidth consumption attacks.
2. Formal game-based modeling and verification of two client puzzle protocols.
3. Formal game-based verification of the defense against resource exhaustion DoS attacks in the JFK key exchange protocol.

**Organization of the paper.** In section 2, we describe related work. Basic client puzzle protocols are described in section 3. We then propose our own distributed client puzzle protocol in section 4. In section 5, we briefly present our formal analysis tools, including ATL (alternating-time temporal logic), its game semantics and the MOCHA finite-state verification tool. We then model the protocols, state their properties in ATL, and use MOCHA to verify them. In section 6, we describe game-based analysis of DoS prevention in the JFKr protocol. Conclusions are in section 7.

## 2. Related work

We first describe the state of the art in denial-of-service prevention mechanisms. We then survey relevant related work on formal methods in security, focusing on, respectively, formal modeling of availability-related security properties, game-based verification of security, and previous formal analyses of the JFK key exchange protocol.

### 2.1 Denial-of-service prevention mechanisms

**Reactive mechanisms.** Development of anti-DoS mechanisms has received enormous attention. Proposed approaches include pushback [19, 28], traceback [37, 38, 39], and filtering [6, 30, 34, 46, 47].

Pushback mechanisms [19, 28] require router support to rate-limit aggregate flows responsible for congestion (an aggregate is defined as a collection of packets that share some common property), and push filters upstream towards the sources of these aggregate flows. Pushback faces challenges in attack traffic identification and ISP co-operation, and requires non-negligible state at the routers.

IP traceback schemes are used to find attacks' origins. Traceback information is marked by routers in the IP identification field. In the Probabilistic Packet Marking Scheme [37, 39], each packet is probabilistically marked with partial path information. The victim can reconstruct the attack paths if it receives a significant number of packets. Another IP traceback scheme, known as hash-based traceback [38], stores packet digests in the form of Bloom

filters [8] at each router. By checking neighboring routers iteratively with attack packets, the attack path can be constructed. The inherent assumption of traceback mechanisms is that they can generate attack signatures and differentiate attack and legitimate traffic. This is a non-trivial problem, however, and traceback approaches are less useful if the attack traffic originates from genuine source addresses and is statistically similar to legitimate traffic. Traceback schemes also suffer from scalability problems.

Andersen proposed Mayday [6] that uses Secure Overlay Services (SOS) [22] architecture to proactively defend against DoS Attacks. Park and Lee proposed a distributed packet filtering (DPF) mechanism [34] against IP address spoofing. DPF relies on BGP routing information to detect spoofed IP addresses. Mirkovic *et al.* [30] proposed detection and filtering close to the attacker. The Pi marking scheme [46] enables the victim to detect packets with a spoofed source IP address. Stateless Internet Flow Filter (SIFF) [47] enables the victim to stop individual traffic flows from reaching it without keeping per-flow state in the network. SIFF defends against bandwidth flooding attacks. Still, both Pi and SIFF require differentiation of legitimate and attack traffic.

The main challenge for reactive mechanisms is the correct differentiation between legitimate and attack traffic. Reactive mechanisms produce non-zero false positives (*i.e.*, legitimate packets are sometimes dropped), which is a DoS issue in itself. Therefore, there is a need for proactive mechanisms that drop only attack packets and ensure that the legitimate traffic always reaches its destination.

**Client puzzles.** Client puzzles are a proactive mechanism to defend against denial of service attacks. They have been proposed in the context of TCP [12, 13, 14, 20, 43, 44], authentication protocols [7, 26], and TLS [11].

The puzzle auctions protocol of [43] allows clients to bid for service by computing puzzles with difficulty levels of their own choice. The server under attack allocates its limited resources to requests carrying the highest priorities. The bidding strategy is such that the client can gradually raise its bid until it wins. Feng has described the importance of implementing the puzzles at the IP layer [12, 13]. Most puzzle-solving techniques concentrate on collecting idle CPU cycles to generate a solution. Abadi *et al.* [2] proposed a memory bound puzzle that imposes a memory access cost upon the client in an effort to impose similar puzzle solving delays on different hardware platforms.

Several papers [14, 21, 33] proposed to use application-layer graphic Turing tests (GTTs) as puzzles to prevent automated flooding. GTTs differentiate human users from automated attack zombies and filter attack traffic at the server. By contrast, we focus on IP-layer puzzles.

### 2.2 Formal models

**Formal models of denial of service.** Meadows [29] (followed by Ramachandran [36]) proposed a cost-based framework for analysis of denial-of-service attacks and used the NRL protocol analyzer to carry out the analysis. Her framework is based on Gong and Syverson's fail-stop model [15]. The approach of [29] is well-suited to resource exhaustion attacks since it models DoS vulnerability as asymmetry of computational costs between participants.

Lafrance and Mullins [25] proposed to detect DoS vulnerabilities in security protocols using the process algebra SPAA and a cost-based framework. They introduce an information flow property, called *impassivity*, which detects when an attacker, using his low-cost actions, causes interference on high-cost actions of other principals. This model also appears to be best suited to resource exhaustion attacks.

Gunter *et al.* [16] presented a mathematical characterization of anti-DoS properties of an authenticated broadcast protocol. By contrast, our paper describes a formal, tool-supported analysis of a different set of anti-DoS guarantees.

**Game-based verification.** Game-based formal analysis frameworks are a good fit for adversarial protocols, in which the main challenge for an honest participant is to defend against strategic behavior by the other party. ATL and MOCHA have been successfully used to analyze adversarial protocols for fair exchange, contract signing, and non-repudiation [9, 24, 23]. To the best of our knowledge, this paper is the first to formally model anti-DoS mechanisms as adversarial protocols, and to apply a game-based framework to their verification.

**Formal analysis of the JFK protocol.** For the purposes of this paper, we are only interested in the DoS prevention mechanisms of the JFK key exchange protocol [3]. The protocol in its entirety has been previously analyzed by Datta *et al.* [10] and Abadi *et al.* [1], but their work is complementary to ours in that they do not attempt to capture the adversarial nature of the protocol in the formal model itself.

## 3 Basic client puzzle protocols

*Client puzzle* protocols require clients to solve puzzles before getting resources from the server. The idea is to force clients to pay (in this case, with their own resources) in order to use network services. In this section, we describe the auction protocol of Wang and Reiter [43], and the challenge-response protocol of Feng *et al.* [13].

### 3.1 Puzzle auctions

As all puzzle-based anti-DoS protocols, the puzzle auctions protocol of Wang and Reiter [43] requires the client

to solve a puzzle of a certain difficulty level before he can initiate a session with the server. The request message must be accompanied by the solution to the puzzle. Puzzles are based on hash reversals. A puzzle includes a nonce parameter $N_s$ created by the server, and a nonce parameter $N_c$ created by the client. The purpose of the nonces is to ensure freshness of the puzzles, thus preventing replay attacks. The solution to the puzzle is the string $X$ such that the first $m$ bits of $h(N_s, N_c, X)$ are zeroes ($h$ is the hash function). The client performs a brute-force search on the value of $X$ until he finds the value that produces the right hash. The difficulty level of the puzzle is determined by the value $m$.

The server adapts the difficulty level $m$ depending upon its current utilization. Given a request message, the server can either accept the request and continue with the protocol, or send a rejection to the client. The latter may then increase the difficulty level of the puzzle and send the request back to the server. Legitimate clients get access to the server by increasing the difficulty level, whereas an attacker will have less incentive to do that because he would not want to alert the zombie machine's owner by excessive computation. Puzzle auction protocols thus ensure that legitimate clients will eventually get access to the server, while attack traffic is filtered out at the server since the attacker can only provide puzzle solutions up to a certain level.

### 3.2 Challenge-response puzzles

In the approach proposed by Feng *et al.* [13], instead of clients increasing the puzzle difficulty level for each server rejection, the server is required to respond with a puzzle of the current highest difficulty level. The server allocates resources only if it receives the correct solution from the client. By adapting the puzzle difficulty level in proportion to the current load, the server can force clients to solve puzzles of varying difficulty.

For game-based modeling, we will use the challenge-response protocol of [13]. A slightly modified version of the protocol is depicted in fig. 1. To understand the protocol, recall the standard TCP session establishment protocol, in which the client starts a new TCP session by sending a SYN request to the server. The server responds with a SYN-ACK message, which the client confirms by returning an ACK message. In the SYN flooding attack, the attacker sends a large number of SYN requests with spurious source addresses. If the server allocates and maintains TCP state for each request in its connection queue, it will soon run out of room to accommodate new requests, and legitimate clients will be denied service.

In the client-puzzle version of the protocol, the client attaches its nonce $N_c$ and a timestamp $TS$ to its SYN request message. Upon receiving the request, the server generates a puzzle, if required (depending upon the current server utilization levels) and sends the puzzle parameters $P$ (including the difficulty level, *i.e.*, the number of missing bits the client is expected to compute), server nonce $N_s$ and the hash value $H = h(N_c, N_s, F, X)$ to the requesting client. $F$ is the flow identifier, defined as the quintuple (source IP address, destination IP address, source port, destination port, and protocol number). Note that in this protocol, as well as in the rest of this paper, the client verifies his puzzle solution by recomputing $H$ and matching it against the hash value received from the server (protocol of section 3.1 uses a different verification procedure).

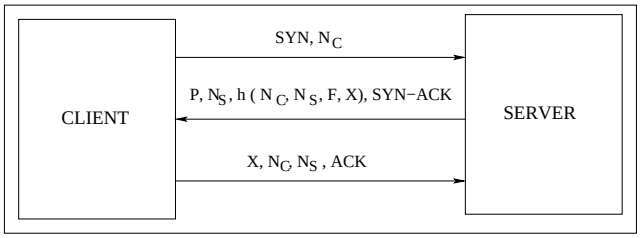


**Figure 1. Challenge-response client puzzles.**

Upon receiving the puzzle, the client computes the solution $X$ by brute-force search, and sends it back to the server. The server verifies the solution and then allocates resources. Because the server does not maintain any state until the solution is verified, SYN flooding attacks are prevented. (It may appear that the server is not truly stateless since it must remember the puzzles it previously generated, but, as we explain in section 4.4, this can be avoided if puzzles are computed as HMAC of connection parameters and server's secret).

### 3.3 Drawbacks of basic client puzzle protocols

Basic client puzzle protocols work well to prevent *resource exhaustion* attacks on servers. They fail, however, if the attackers direct a huge surge of traffic to a particular server, flooding that server's bandwidth and causing the server to start dropping packets. We will refer to this attack as the *bandwidth consumption* attack. To block this kind of malicious traffic, the filtering mechanism must be deployed in the intermediate network rather than at the server level. In section 4, we propose a new distributed client puzzle protocol that can prevent bandwidth flooding attacks.

## 4 Distributed client puzzle system

### 4.1 Overview of the system

In order to block bandwidth-consumption traffic early on, we propose a new *distributed client puzzle protocol* (or,

more precisely, a suite of protocols) that moves puzzle generation and verification from the servers to the network. Our protocols employ standard hash-reversal puzzles.

Our system comprises a set of monitoring and filtering nodes that form an overlay network to detect and filter attack traffic. The monitoring nodes continuously monitor traffic to detect anomalies such as SYN floods and unusual inflections in the incoming and outgoing traffic rates. Once an anomaly in traffic to a particular server is detected, the monitoring nodes send BGP route advertisements to the access (edge) routers, claiming that the filtering nodes have the shortest path to the server in question. This will result in all incoming traffic destined to this server being diverted to the filtering nodes. The routing tables of core routers are not changed. Loops are avoided since edge routers can differentiate the traffic entering and leaving the network. Only incoming traffic is diverted to the filters. This re-routing mechanism works well both within a single ISP and across multiple ISPs.

The filtering nodes use client puzzles to block attack traffic and allow the legitimate traffic to go through to the server. Once the attack subsides, the monitoring nodes can issue BGP route updates to revert the traffic flow back to its regular path.

A client who wants his request to reach the server must solve the puzzle presented by the filtering node. Client functionality remains the same as in the previously proposed client-puzzle protocols. It is important that the monitoring nodes detect the unusual volume of traffic at an early stage of the attack and inform the filtering nodes of the proper puzzle difficulty level. The monitoring nodes have to measure server flows (*i.e.*, all traffic going to a particular destination address and port), and the puzzle distribution mechanism must be applied on the per-flow basis, with the puzzle difficulty level proportional to the flow volume. This means that different clients communicating with the same server, possibly through different filtering nodes, must be solving puzzles of the same difficulty level.

Our distributed protocol comprises three main parts: **distributed network monitoring** to obtain a global view of traffic statistics; **puzzle distribution** to present puzzles to clients and adaptively vary their difficulty; **distributed filtering** to filter flows based on the results of puzzle-solving by clients.

We give a semi-formal specification of main protocols in fig. 3 and 4.

### 4.2 Distributed monitoring

We will assume a set of monitoring nodes deployed in the intermediate network in such a way that any packet traversing the network passes through exactly one node. For the purposes of DDoS detection, we define a flow as

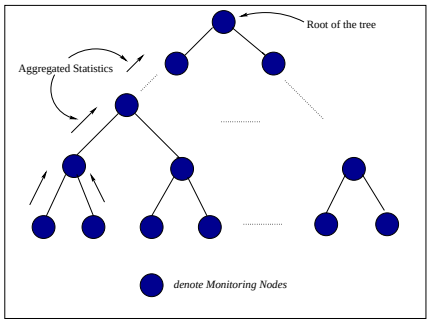


**Figure 2. Hierarchical deployment of monitor nodes for distributed network monitoring.**

⟨destination IP address, destination port number⟩. In general, a flow is some $k$-tuple $F$ (for example, for worm detection the flow should be defined by the source IP address and destination port).

Every measurement interval, the monitoring nodes record the per-flow traffic information (packet and byte count) into a table. A counting Bloom filter may be used to reduce storage requirements. At the end of the measurement interval, the monitoring nodes communicate and exchange the flow-level statistics to gather the global view of traffic. For this communication, we organize our monitoring nodes into a hierarchical tree structure. This is depicted in fig. 2. Left and right child nodes send traffic information to their parents, where the information is aggregated and then forwarded up the tree until it reaches the root. At the root node, aggregated packet and byte information are used to analyze traffic behavior. A large number of packets associated with a low number of bytes indicates a probable DDoS attack. A large number of bytes may indicate a flash crowd (sudden surge of legitimate traffic) or an attack.

Note that we do *not* use this classification to separate traffic into "good" and "bad." Instead, we use it to generate client puzzles of different difficulty levels and let the clients' puzzle-solving behavior clarify their intent. Attackers will not be able to solve the presented puzzles, and their traffic will be dropped by the filtering nodes.

### 4.3 Puzzle distribution

Once the root of the hierarchy has computed the aggregated flow information, it determines which flows exceed the *global* threshold(s), as well as the destination addresses of these flows. Multiple global thresholds are used to determine the puzzle difficulty levels. Different flows may have different traffic statistics, and puzzle difficulty levels are proportional to the total per-flow traffic. If the traffic for a given flow varies across measurement intervals, the puz-

zle difficulty level would adapt depending upon the current traffic (this is done at the end of each measurement interval). For instance, for traffic counts between $(T_1, T_2)$, the difficulty level is $D_1$, between $(T_2, T_3)$, the difficulty level is $D_2$, and so on, where $T_1 < T_2 < T_3 < \ldots$ are different global thresholds, and $D_1$ is the lowest puzzle difficult level.

If the traffic for at least one flow exceeds the least global threshold, the root node issues BGP advertisements to edge routers, instructing them to divert all traffic destined to the corresponding server to the nearest filtering node. The aggregated per-flow volume information is used to derive the proper puzzle difficulty level for clients of each identified server. These per-flow puzzle difficulty levels are communicated to the filtering nodes. Note that, from the client's viewpoint, the difficulty of the puzzle depends on the destination. In particular, access to destinations that are *not* under attack does not require clients to solve difficult puzzles.

### 4.4 Distributed filtering

Once the filtering nodes have set the puzzle difficulty level for each destination under attack and traffic to these destinations has been diverted to the filtering nodes, clients are forced to solve puzzles of the appropriate difficulty level. This ensures that traffic from zombie machines does not reach the end-link to the destination server. We assume that, for each puzzle difficulty level, the filtering node maintains a "white list" of source IP addresses that have solved a puzzle of that level. This ensures that a client needs to prove its legitimacy only once. We also assume that all filtering nodes share the same secret key to be used in puzzle generation, and that their clocks are loosely synchronized.

If client and server belong to different ISPs, client's packets may traverse multiple filtering nodes on their way to the destination. To ensure that the client needs to solve a puzzle only once, we take the following approach (described below for the specific case of a SYN flooding attack, but easily adaptable to any bandwidth flooding attack).

When a filtering node sees a SYN packet, it first checks whether the source is in the white list associated with the puzzle difficulty level required by the destination server. If not, the filter generates a puzzle (as HMAC of the connection parameters and nonces, with the filters' common secret as the key) and sends it along with SYN-ACK to the client. The filter does not forward the SYN to the server, and SYN-ACK is spoofed to look as if it is coming from the server.

Once a solution and ACK are received from the client, the filtering node re-generates the puzzle and verifies the solution. If verification succeeds, then the source address is added to the white list, and the solution and ACK are forwarded to the next-hop router. If there is another filtering node in the routing path, it will assume that this ACK has been sent in response to some previously generated puzzle (filtering nodes are stateless, and don't remember the puzzles they generate). It will again verify the solution. Assuming not too much time has passed, verification will succeed since this node will be using the same timestamp and the same secret key as the filter that originally generated the puzzle.

Once the ACK packet with the puzzle solution reaches the server, there are two possibilities. If the server's TCP stack has been modified to make it stateless and to verify client puzzles (with the same key that the filtering nodes are using), then the server will accept the connection request. If the server is running a normal TCP stack, then it will drop the packet or send RST to the client since he has not seen the corresponding SYN packet. The client will time out, and send a second SYN packet to the server. By now, all filtering nodes on the routing path have this client in their white lists and hence forward its SYN packet without any interference, resulting in the establishment of the normal TCP connection between the client and the server. Fig. 5 depicts what happens when the client's connection request traverses multiple filtering nodes along the route to the server.

Note that white lists of client IP addresses are maintained at the filtering nodes for each puzzle difficulty level, *not* for each destination address. A client in the white list for a particular difficulty level is automatically added to the white lists for all lower difficulty levels. The white lists are flushed after a time interval (which could be as long as 30 minutes). This is done to ensure that an attacker does not launch a DDoS attack by compromising legitimate nodes over time after the latter have solved difficult puzzles.

**Replay attack on stateless filters.** In our solution, filtering nodes responsible for generating puzzles are stateless. In particular, it is not necessary for them to remember every puzzle they sent to clients (otherwise, they would also be vulnerable to a flooding denial-of-service attack). Instead of randomly generating a new puzzle value $X$, the filtering node can compute it as keyed hash (*e.g.*, HMAC) of the connection parameters, nonces, and a (coarse) timestamp, using a secret key which is known only to the filtering nodes (same key for all nodes). This guarantees that $X$ appears (pseudo-)random to the client, yet the node does not need to store $X$ since it can be re-computed whenever the client presents his ostensible solution. The client must respond within the same coarse time slice, or else the timestamp will change, and the node will not accept the client's value of $X$.

One potential drawback of this approach is the possibility of replay of old puzzle solutions within a *single* time slice. The scope of this threat is limited. Old solutions become invalid as soon as the timestamp changes, and the attacker needs to know the exact connection parameters and nonces of a legitimate connection, which is typically not

**Traffic monitoring at each monitor node:**

1. **for** each incoming packet **do**
2. flow definition $F$ = <dest_IP, dest_port>;
3. update local counting Bloom filter;
4. **if** end of measurement interval **then**
5. send traffic information (counting Bloom filters) along the hierarchy of monitor nodes to the root;
6. **fi**
7. **rof**

**Traffic aggregation at the root node:**

1. aggregate traffic statistics obtained from all monitor nodes;
2. determine the mapping of aggregated counts to different global thresholds;
3. determine the appropriate puzzle difficulty level;
4. activate the filtering nodes and inform them of the puzzle difficulty levels for each destination;
5. divert routing tables by sending BGP route advertisements to the edge routers.

**Figure 3. Protocol specification: distributed monitoring and puzzle distribution.**

**Filtering at each filter node:**

1. **for** each incoming packet **do**
2. **if** the source IP belongs to the white list **then**
3. forward the packet along the route to server;
4. **else**
5. **if** packet contains the correct puzzle solution **then**
6. add source IP to white list and forward packet;
7. **else**
8. **if** packet does not contain solution **then**
9. generate and send a puzzle to the client;
10. **fi**
11. **if** packet contains incorrect puzzle solution **then**
12. drop the packet;
13. **fi**
14. **fi**
15. **fi**
16. **rof**

**Figure 4. Protocol specification: distributed filtering.**

the case in flooding attacks. We also note that even successful replay does not result in allocation of new filtering node or server resources since all replayed puzzle solutions refer to the same client-server connection (although short-term bandwidth exhaustion is a possibility).

One possible defense against replay is to cache *correctly solved* puzzles at the filtering node. This would require

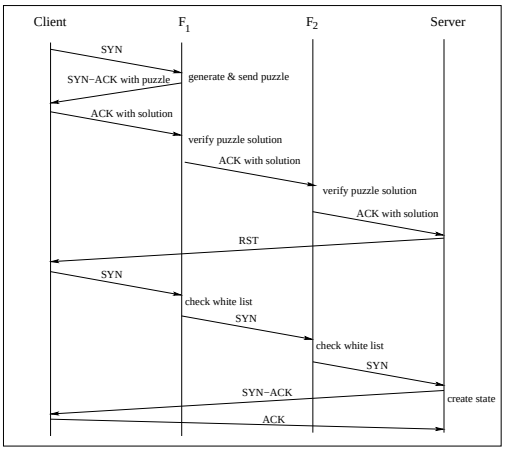


**Figure 5. Legitimate traffic traverses multiple filtering nodes. The figure depicts the scenario with two filtering nodes $F_i$ along the path from client to server. The client is legitimate and the first filtering node $F_1$ verifies the puzzle solution and sends ACK with solution to the next node. $F_2$ also verifies the solution, and does not force the client to solve a new puzzle.**

the node to maintain some state, but only for "white-listed" connections in which the client has successfully solved the puzzle. This technique is borrowed from the JFK protocol, described below in section 6, except that in JFK server cookies rather than puzzles are cached. The objective of server cookies in JFK is simply to confirm that the client is listening at the ostensible source IP address, while the objective of client puzzles is both to confirm the address and to force the client to perform a computationally intensive task.

**Adaptive traffic variation attack.** Our proposed system is potentially vulnerable to an attack in which the attacker uses the measurement interval to launch large volumes of traffic. At the end of the interval, the traffic surge will be detected, and the filtering nodes will be activated to stop attack traffic. Because the attacker can no longer solve difficult puzzles being presented to him, he stops sending packets. At the next interval, since the attack has subsided, the packet flow reverts back to its normal path. The attacker can now re-launch its attack and continue doing so every other measurement interval. This is a difficult attack to defend against, and we conjecture that any anti-DoS system based on adaptive traffic measurement will be vulnerable. At the very least, our proposed system will defend against non-adaptive attacks such as those launched by automated scripts and random-scanning worms.

### 4.5 Example

To illustrate our protocol with an example, we describe a small configuration, similar to the one we will later analyze using MOCHA. We assume that 3 clients are sending traffic to a destination server $S$ via monitoring nodes $M_1$, $M_2$, and $M_3$. Traffic on a link from client $i$ to monitor $j$ is given by $t^i_j$. Therefore, the difficulty level of the puzzle for every client (since all clients transmit to the same destination) is collaboratively computed by monitoring nodes $M_1$, $M_2$, and $M_3$ as $D = \sum_{1 \leq i \leq 3, 1 \leq j \leq 3} t^i_j$.

### 4.6 Desired security properties

We now describe the properties that must be maintained by a DDoS prevention protocol. In section 5, they will be formalized in alternating-time temporal logic.

**Availability** The DDoS prevention protocol should ensure that legitimate clients are always guaranteed access to server resources (provided they solve puzzles of the appropriate difficulty level). Any coalition of attackers should not be able to prevent the legitimate clients from accessing the service.

**Liveness** If the server has enough resources to handle connection requests, then any requesting client (legitimate or malicious) should obtain access to the server.

**Client authentication** Clients that do not solve the puzzle should not be allocated resources by the server.

**Adaptability** Puzzle difficulty level should be proportional to the amount of traffic going to the server, and it should adapt depending upon the current utilization levels at the server.

## 5 Game-based model

In this section, we discuss our formal modeling and verification of the denial-of-service prevention protocols. Protocols are specified as alternating transition systems (ATS), their properties are stated in alternating-time temporal logic (ATL) and verified using the MOCHA model checker [5]. ATS and ATL were originally proposed by Alur *et al.* [4]. Our modeling techniques are similar to those previously used by Kremer *et al.* for fair exchange protocols [9, 24, 23], but the set of protocols and properties we consider are substantially different.

As in previous work on game-based verification of security protocols, we only verify a small finite configuration of the system. The ATS/MOCHA formalism cannot be used (at least not directly) to prove that a protocol is correct regardless of the size of configuration. Formally proving that our system guarantees availability and adaptability for configurations of arbitrary size is an interesting topic of future research.

### 5.1 ATS, ATL, and MOCHA

To capture the adversarial nature of denial-of-service prevention protocols, we model protocols as alternating transition systems (ATS) [4], which are a game variant of Kripke structures. An ATS is composed of a set of players $\Sigma$, a set of states $Q$ that represent all possible game configurations, a set $Q_0 \subseteq Q$ of initial states, a set of propositions $\Pi$, a labeling function $\pi : Q \rightarrow 2^{\Pi}$ that labels each state with a set of propositions true in the state, and a game transition function $\delta : Q \times \Sigma \rightarrow 2^{2^Q}$.

For a player $a$ and a state $q$, $\delta(q, a)$ is the set of choices that $a$ can make in the state $q$. A choice is a set of possible next states. One step of the game at state $q$ is played as follows: each player $a \in \Sigma$ makes its choice and the next state of the game $q'$ is the intersection of the choices made by all the players of $\Sigma$, *i.e.*, $\{q'\} = \cap_{a \in \Sigma} \delta(q, a)$. A computation is an infinite sequence $\lambda = q_0 q_1 ... q_n ...$ of states obtained by starting the game in $q_0 \in Q_0$.

Alternating-time temporal logic (ATL) is defined with respect to a finite set $\Pi$ of propositions and a finite set $\Sigma$ of players. For a set of players $A \subseteq \Sigma$, a set of computations $\Lambda$, and a state $q$, consider the following game between a "protagonist" and an "antagonist" starting in $q$. At each step, to determine the next state, the protagonist selects the choices controlled by the players in the set $A$, while the antagonist selects the remaining choices. If the resulting infinite computation belongs to the set $\Lambda$, then the protagonist wins. If the protagonist has a winning strategy, we say that the ATL formula $\ll A \gg \Lambda$ is satisfied in state $q$. Here, $\ll A \gg$ is a path quantifier, parameterized by the set $A$ of players, which ranges over all computations that the players in $A$ can force the game into, irrespective of how the players in $(\Sigma \setminus A)$ proceed. Details of ATS and ATL can be found in [4].

Instead of modeling protocols directly with ATS, we follow Kremer and Raskin [24] in using a Dijkstra-style guarded command language (details of the language can be found in [17]). Each player $a \in \Sigma$ is associated with a set of guarded commands of the form $guard \rightarrow update$. A computation step is defined as follows: each player chooses one of its commands whose boolean guard evaluates to true, and the next state is obtained by taking the conjunction of the effects of the *update* parts of the commands selected by the players. Given an ATS described in terms of guarded commands, the finite-state verification tool MOCHA [5] automates the job of model checking ATL formulae over

the specified ATS. Excerpts from our MOCHA code can be found in the appendix.

## 5.2 Protocol modeling

### 5.2.1 Challenge-response client puzzle protocol

This protocol was originally proposed by Feng *et al.* [13] (see section 3.2). For finite-state modeling, we choose a configuration with two legitimate clients and two attackers trying to access the server that can allocate resources for at most two requesting entities. The client is modeled as an entity that can solve puzzles of a high difficulty level, whereas the attackers can only solve puzzles below a certain difficulty level. When all server resources have been allocated and a new entity requests a resource, the server adapts the difficulty level for the requesting entity and, if it receives a solution for this new, difficult puzzle, then the server evicts the client who had previously solved a puzzle of a low difficulty level. If the new client was unsuccessful in solving the difficult puzzle, then the server drops the request.

### 5.2.2 Distributed client puzzle protocol

For our own distributed protocol (see section 4), we model a finite configuration consisting of one client, one attacker, two intermediate filtering nodes, and one server. After some communication (sharing of per-flow summary traffic information), the monitoring nodes determine the puzzle difficulty level for a given flow. Adaptability property (see section 4.6) requires that clients should not be able to get service from the server by solving a puzzle of lower difficulty level than that associated with the total traffic directed to that server, as viewed by the intermediate filtering nodes.

## 5.3 Modeling of anti-DoS properties

In this section, we show how the security properties that a DDoS prevention protocol is required to provide (see an informal description in section 4.6) can be stated as game strategies for different players and easily formulated as ATL formulas. One of the major advantages of using a game-based framework is that it enables us to directly and formally model adversarial and cooperative behaviors between different entities taking part in a protocol run. For example, we can formally model the collaborative strategy of a coalition of filtering nodes that cooperate in order to defeat a coalition of malicious clients.

### 5.3.1 Properties of challenge-response client puzzles

The ATL formulae that must be maintained by the protocol are given below. In the formulae, $Clients$ are legitimate clients, $Servers$ are the servers and $full$ is the state predicate stating that all resources at the server are allocated. Boolean variables $requestClient$ and $requestAttacker$ indicate whether the entity sending a request to the server is legitimate or malicious. Boolean variables $allocatedClient$ and $allocatedAttacker$ indicate whether the server allocated resources in response to a request from a legitimate (respectively, malicious) client. Boolean variables $ClientPuzzleSolution$ and $AttackerPuzzleSolution$ indicate whether the client (respectively, attacker) has correctly solved the puzzle.

Note that a property of the form $\ll X \gg \phi$ does *not* mean that only members of $X$ are participating in the protocol. For example, the client authentication property says that, regardless of the attacker's actions, the coalition of honest clients and servers has a strategy such that in any state in which the attacker makes the predicate $AttackerPuzzleSolution$ (over attacker's state variables) false, the predicate $allocatedAttacker$ (over servers' state variables) is also false, due to previous actions by clients and servers.

**Availability** Expressed by this ATL path formula:

$$\ll Clients, Servers \gg \\ \Diamond((ClientPuzzleSolution \rightarrow allocatedClient) \wedge \\ (allocatedClient \rightarrow ClientPuzzleSolution))$$

This says that the server allocates resources to a client if and only if the client has solved the puzzle presented to him. There is no adaptation of the puzzle difficulty level in the basic challenge-response client puzzle protocol.

**Liveness** Expressed by this ATL path formula:

$$\ll Servers \gg \\ \Diamond((\neg full) \rightarrow \\ (requestClient \rightarrow allocatedClient) \wedge \\ (requestAttacker \rightarrow allocatedAttacker))$$

This says that in any state in which the server is not full, it grants resources to all requesters regardless of whether they are legitimate or malicious.

**Client authentication** Expressed by this formula:

$$\ll Clients, Servers \gg \\ \Diamond(\neg AttackerPuzzleSolution \wedge full \rightarrow \\ \neg allocatedAttacker)$$

This says that legitimate participants have a strategy that results in not allocating resources to an attacker who has not solved the puzzle presented to him.

We used the MOCHA model checker to formally verify that the above three properties hold for the basic challenge-response client puzzle protocol of Feng *et al.* [13].

#### 5.3.2 Properties of distributed client puzzles

In the ATL formulae below, $Clients$ are legitimate clients, $Servers$ are the servers, $Filters$ are the intermediate filtering nodes, $Monitors$ are the intermediate monitoring nodes and $full$ is the state predicate stating that all resources at the server have been allocated. Boolean variables $requestClient$ and $requestAttacker$ indicate whether the entity sending a request to the server is legitimate or malicious. Boolean variables $allocatedClient$ and $allocatedAttacker$ indicate whether the server allocated resources in response to a request from a legitimate (respectively, malicious) client. Boolean variables $ClientPuzzleSolution$ and $AttackerPuzzleSolution$ indicate whether the puzzle has been solved correctly, $difficulty\_level$ is the current puzzle difficulty level, and $pkts1[0]$ and $pkts1[1]$ contain the number of packets from client 1 to monitor 0 (monitor 1, respectively). Since we are only modeling one server, that server is assumed to be the destination of all the packets.

**Availability** Expressed by this ATL path formula:

$$\ll Clients, Filters \gg \Diamond((ClientPuzzleSolution \rightarrow allocatedClient) \wedge (allocatedClient \rightarrow ClientPuzzleSolution))$$

This is similar to the property for the basic challenge-response protocol, except that here it's the *intermediate filtering nodes* (possibly in coalition with legitimate clients) who must have a collaborative strategy for ensuring that server resources are allocated to a client if and only if the client has solved the puzzle presented to him.

Note that while the availability property *per se* does not model the increasing difficulty of the puzzles as the traffic volume increases, the *adaptability* property (see below) guarantees that the current difficulty level is equal to the total traffic volume for a given flow.

**Liveness** Expressed by this ATL path formula:

$$\ll Filters, Servers \gg \Diamond((\neg full) \rightarrow (requestClient \rightarrow allocatedClient) \wedge (requestAttacker \rightarrow allocatedAttacker))$$

Same as for the basic protocol, except that servers are assisted by filtering nodes in ensuring that, provided the server is not full, everybody is granted access.

**Client authentication** Expressed by essentially the same formula as for the basic challenge-response protocol:

$$\ll Clients, Filters \gg \Diamond(\neg AttackerPuzzleSolution \wedge full \rightarrow \neg allocatedAttacker)$$

**Adaptability** Expressed by this ATL path formula:

$$\ll Filters, Monitors \gg \Diamond(difficulty\_level = pkts1[0] + pkts1[1])$$

This says that the filtering and monitoring nodes have a collaborative strategy for reaching a state in which the difficulty level of puzzles is equal to the total traffic (across both monitors) directed to the destination server.

We used the MOCHA model checker to formally verify that all four properties hold for our distributed client puzzle protocol.

## 6 DoS prevention for key establishment

In this section, we formalize one of the variants of the Just Fast Keying (JFK) key establishment protocol [3], and use the MOCHA model checker to verify its resistance to resource exhaustion attacks.

### 6.1 JFKr protocol

The JFK design paper [3] describes two variants of the protocol. The variants are very similar. JFKr protects the responder's identity against active attacks and the initiator's identity against passive attacks, while JFKi protects the initiator's identity only against active attacks. Both variants aim to defend the responder against resource exhaustion attacks caused by a malicious client or clients initiating a large number of spurious protocol sessions.

We limit our attention to the JFKr variant. Our notation appears in fig. 6.

**JFKr specification.** The JFKr protocol consists of four messages. The third and fourth messages are protected using keys $K_a$ (for integrity checking) and $K_e$ (for encryption). The keys are computed as $K_{a,e} = hash_x(N_i, N_r, \{\text{“a”}, \text{“e”}\})$, where $x$ is the shared secret computed as the joint Diffie-Hellman value: $x = x_i^{d_r} = x_r^{d_i}$. Here $d_{r,i}$ is the responder's (respectively, initiator's) DH exponent (see fig. 6).

$$\begin{array}{ll} I \rightarrow R: & N_i, x_i \\ R \rightarrow I: & N_i, N_r, x_r, g_r, t_r \\ & \text{where } t_r = hash_{K_r}(x_r, N_r, N_i, IP_i) \\ I \rightarrow R: & N_i, x_i, N_r, x_r, t_r, e_i, h_i \\ & \text{where } e_i = enc_{K_e}(ID_i, ID_r^{'}, sa_i, \\ & \qquad sig_{K_i}(N_r, N_i, x_r, x_i, g_r)), \\ & \quad h_i = hash_{K_a}(\text{“i”}, e_i) \\ R \rightarrow I: & e_r, h_r \\ & \text{where } e_r = enc_{K_e}(ID_r, sa_r, \\ & \qquad sig_{K_r}(x_r, N_r, x_i, N_i)), \\ & \quad h_r = hash_{K_a}(\text{“r”}, e_r) \end{array}$$

| | |
|---|---|
| $hash_k(M)$ | Keyed hash of message $M$ using key $k$. |
| $enc_k(M)$ | Encryption of message $M$ using symmetric key $k$. |
| $sig_{k_i}(M)$ | Digital signature of message $M$ under principal $i$'s private key. |
| $N_i$ | Initiator's nonce. |
| $N_r$ | Responder's nonce. |
| $d_i$ | Initiator's secret exponent. |
| $d_r$ | Responder's secret exponent. |
| $g$ | Diffie-Hellman group generator. |
| $x_i = g^{d_i}$ | Initiator's Diffie-Hellman value. |
| $x_r = g^{d_r}$ | Responder's Diffie-Hellman value (same for multiple sessions). |
| $IP_i$ | Initiator's IP address. |
| $IP_r$ | Responder's IP address. |
| $sa_i$ | Initiator's desired security association. |
| $sa_r$ | Security association that the responder may need to give to the initiator (*e.g.*, responder's SIP in IPSec). |

**Figure 6. Notation for the JFKr model.**

### 6.2 Modeling of anti-DoS properties

The JFK protocol employs a cookie-based scheme for preventing resource consumption attacks. After the first message, the responder computes a *cookie* (an unforgeable keyed hash value of the information identifying the connection) and sends it back to the initiator ($t_r$ value in the specification of section 6.1). The responder does not establish any state until the initiator returns the cookie in the third message of the protocol. The responder verifies the cookie by recomputing the keyed hash and comparing it with the value returned by the initiator. Verified cookies are cached at the responder to prevent the replay of old cookies. This cache grows with time as the number of legitimate connections increases, and is flushed when the responder's secret changes, invalidating all existing cookies. In our simplified attacker model, we do not attempt to model cookie replay attacks, and thus our model for the JFK responder does not include the cookie cache (see [1] for a full-fledged model of the JFK protocol).

Unlike the client puzzle protocols considered in the rest of this paper, the purpose of responder cookies in JFK is simply to confirm that the source IP address is legitimate, while the purpose of client puzzles is both to confirm the client address and to force the client to perform a computational intensive task such as guessing the missing bits in the cookie (puzzle).

In the formulas below, $responder\_cookie$ is a predicate which holds iff the initiator returned the correct cookie, and $hash_i$ holds iff the value of the initiator's MAC $h_i$ is equal to that computed by the server. Flag $fail\_stop\_Responder$ is used to indicate that the responder has stopped execution of the protocol before it could complete. This follows Gong and Syverson's fail-stop model [15].

Before completing the protocol and starting to use the established key, the initiator is expected to verify the MAC (hash value) sent by the responder in the fourth message of the protocol. Let $hash_r$ be the predicate which holds iff this value matches that expected by the initiator.

**Fail-stop responder** Expressed by this ATL path formula:

$$\ll Responder \gg \Diamond((\neg responder\_cookie \vee \neg hash_i) \rightarrow fail\_stop\_Responder)$$

If the responder does not receive the cookie or the cookie is not equal to the expected hash value, then the responder may abort the protocol.

**Fail-stop initiator** Expressed by this ATL path formula:

$$\ll Initiator \gg \Diamond((\neg hash_r) \rightarrow fail\_stop\_Initiator)$$

If the responder's hash value is incorrect, then the initiator will abort the protocol.

## 7 Conclusions

We presented a new protocol that protects against malicious bandwidth consumption by using adaptive client puzzles and pushing their generation to the intermediate routers (rather than destination servers). The routers adaptively change the difficulty level of the puzzles depending on the global measurement of flows directed to a particular destination.

We also demonstrated how game-based verification techniques can be used to analyze availability-related properties of network protocols. As in the case of other adversarial protocols such as fair exchange, alternating-time temporal logic provides a concise and powerful formal language for expressing the properties of interest.

Our case studies include two client puzzle protocols and a state-of-the-art key establishment protocol. Their anti-DoS properties have been verified automatically using the MOCHA model checker. We believe that game-based formal methods are a natural fit for verification problems arising in the analysis of availability, and can be applied to a wide range of denial-of-service prevention protocols.

**Acknowledgments.** We are very grateful to the anonymous CSFW reviewers for their insightful comments. Their suggestions have significantly improved our protocols.

## References


[1] M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the Pi calculus. In *Proc. ESOP '04*, pages 340–354, 2004.

[2] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *Proc. NDSS '03*, pages 25–39, 2003.

[3] W. Aiello, S. Bellovin, M. Blaze, R. Canetti, J. Ioannidis, A. Keromytis, and O. Reingold. Just Fast Keying: key agreement in a hostile Internet. *ACM Trans. Information and Systems Security*, 7(2):242–273, 2004.

[4] R. Alur, T. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[5] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In *Proc. CAV '98*, pages 512–525, 1998.

[6] D. Andersen. Mayday: distributed filtering for Internet services. In *Proc. USENIX Internet Technologies and Systems '03*, 2003.

[7] T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In *Proc. 8th Security Protocols Workshop*, pages 170–178, 2000.

[8] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[9] R. Chadha, S. Kremer, and A. Scedrov. Formal analysis of multi-party contract signing. In *Proc. CSFW '04*, pages 266–279, 2004.

[10] A. Datta, J. Mitchell, and D. Pavlovic. Derivation of the JFK protocol. Kestrel Institute Technical Report KES.U.02.03, July 2002.

[11] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proc. USENIX Security '01*, pages 1–8, 2001.

[12] W. Feng. The case for TCP/IP puzzles. In *Proc. SIGCOMM Workshop on Future Directions in Network Architecture*, pages 322–327, 2003.

[13] W. Feng, E. Kaiser, W. Feng, and A. Luu. The design and implementation of network puzzles. In *Proc. INFOCOM '05*, 2005.

[14] V. Gligor. Guaranteeing access in spite of service-flooding attacks. In *Proc. Security Protocols Workshop*, 2003.

[15] L. Gong and P. Syverson. Fail-stop protocols: an approach to designing secure protocols. In *Proc. DCCA '95*, pages 44–55, 1995.

[16] C. Gunter, S. Khanna, K. Tan, and S. Venkatesh. DoS protection for reliably authenticated broadcast. In *Proc. NDSS '04*, 2004.

[17] T. Henzinger, R. Majumdar, F. Mang, and J.-F. Raskin. Abstract interpretation of game properties. In *Proc. SAS '00*, pages 220–239, 2000.

[18] A. Hussain, J. Heidemann, and C. Papadopoulos. A framework for classifying denial of service attacks. In *Proc. SIGCOMM '03*, pages 99–110, 2003.

[19] J. Ioannidis and S. Bellovin. Pushback: router-based defense against DDos attacks. In *Proc. NDSS '02*, pages 79–86, 2002.

[20] A. Juels and J. Brainard. Client puzzles: a cryptographic defense against connection depletion. In *Proc. NDSS '99*, pages 151–165, 1999.

[21] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proc. NSDI*, 2005.

[22] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proc. SIGCOMM '02*, pages 61–72, 2002.

[23] S. Kremer and J.-F. Raskin. Game analysis of abuse-free contract signing. In *Proc. CSFW '02*, pages 206–220, 2002.

[24] S. Kremer and J.-F. Raskin. A game-based verification of non-repudiation and fair exchange protocols. *J. Computer Security*, 11(3):399–430, 2003.

[25] S. Lafrance and J. Mullins. Using admissible interference to detect denial of service vulnerabilities. In *Proc. IWFM '03*, 2003.

[26] J. Leiwo, T. Auro, and P. Nikander. Towards network denial of service resistant protocols. In *Proc. SEC '00*, pages 301–310, 2000.

[27] P. Liu and W. Zang. Incentive-based modeling and inference of attacker intent. In *Proc. CCS '03*, pages 179–189, 2003.

[28] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *ACM CCR*, 32(3):62–73, 2002.

[29] C. Meadows. A cost-based framework for analysis of denial of service in networks. *J. Computer Security*, 9(1/2):143–164, 2001.

[30] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *Proc. ICNP '02*, pages 312–321, 2002.

[31] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an Internet worm. In *Proc. 2nd Internet Measurement Workshop*, pages 273–284, 2002.

[32] D. Moore, G. Voelker, and S. Savage. Inferring Internet denial-of-service activity. In *Proc. USENIX Security '01*, pages 9–22, 2001.

[33] W. Morein, A. Stavrou, C. Cook, A. Keromytis, V. Misra, and R. Rubenstein. Using graphic Turing tests to counter automated DDoS attacks against web servers. In *Proc. CCS '03*, pages 8–19, 2003.

[34] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *Proc. SIGCOMM '01*, pages 15–26, 2001.

[35] V. Paxson. An analysis of using reflectors for distributed denial-of-service attacks. *ACM CCR*, 31(3):38–47, 2001.

[36] V. Ramachandran. Analyzing DoS-resistance of protocols using a cost-based framework. Yale Technical Report YALEU/DCS/TR-1239, July 2002.

[37] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proc. SIGCOMM '00*, pages 295–306, 2000.

[38] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Strayer. Hash-based IP traceback. In *Proc. SIGCOMM '01*, pages 3–14, 2001.

[39] D. Song and A. Perrig. Advanced and authenticated marking schemes for IP traceback. In *Proc. INFOCOM '01*, pages 878–886, 2001.

[40] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In *Proc. USENIX Security '02*, pages 149–167, 2002.

[41] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *Proc. EUROCRYPT '03*, pages 294–311, 2003.
[42] H. Wang, D. Zhang, and K. Shin. Detecting SYN flooding attacks. In *Proc. INFOCOM '02*, pages 1530–1539, 2002.
[43] X. Wang and M. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *Proc. IEEE Security and Privacy '03*, pages 78–92, 2003.
[44] B. Waters, A. Juels, J. Halderman, and E. Felten. New client puzzle outsourcing techniques for DoS resistance. In *Proc. CCS '04*, pages 246–256, 2004.
[45] J. Xu and W. Lee. Sustaining availability of Web services under distributed denial of service attacks. *IEEE Trans. Computers*, 52(2):195–208, 2003.
[46] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDos attacks. In *Proc. IEEE Security and Privacy '03*, pages 93–109, 2003.
[47] A. Yaar, D. Song, and A. Perrig. SIFF: A stateless Internet flow filter to mitigate DDos flooding attacks. In *Proc. IEEE Security and Privacy '04*, pages 130–146, 2004.

# A MOCHA models

**Model for the basic challenge-response client puzzle protocol.** We modeled the protocol of Feng *et al.* [13] with two legitimate clients and two attackers requesting service from a server that can allocate resources to at most two clients. We assume that legitimate clients can solve puzzles of a higher difficulty level than attackers. Therefore, we assign a high priority for puzzle solving to legitimate clients and a low priority to the attackers.

Excerpts from our model are in figs. 7 and 8. Variables are explained below:

**Ci, Ai, S** Legitimate Clients, Attackers, and the Server.

**requesti** Boolean variable indicates request sent to the server.

**allocatedi** Boolean variable indicates whether the server has allocated resources to $i$.

**Xi** Boolean variable indicates puzzle solution from $i$.

**Nc,Ns,Na** Client's (respectively, server's, attacker's) nonce, modeled as a counter.

**Pci** Puzzle sent by the server to the client $i$.

**h_priority** Boolean variable indicates the requesting entity's puzzle solving capability ("true" for clients, "false" for attackers).

**sum** Variable that keeps track of resources allocated at the server.

**Model for the distributed client-puzzle protocol.** The model for our own distributed client puzzle protocol comprises two parts: (a) the challenge-response puzzle protocol between clients and filters (same as for the basic protocol), and (b) inter-monitor node communication that adaptively determines the global client puzzle difficulty level.

For verification of our protocol, we assume filtering and monitoring are performed at the same node. We call such node a monitor. An excerpt of the monitor model appears in fig. 10 (the model for clients, attackers, and servers is the same as in figs. 7 and 8). Variables are explained below:

**Excerpt: Genuine client in the challenge-response client puzzle protocol**

type my_int : (0..2)
private currentS : my_int
**init**
[] true → Nc' := 0; request' := false; h_priority' := true; X' := false; ACK' := false; STOPc' := false; currentS' := 0
**update**
[] ∼ STOPc → request' := true; Nc' := Nc + 1
[] Pc & (Ns > currentS) & hc & ∼ STOPc → X' := true; currentS' := Ns
[] SYNACK & ∼ STOPc → ACK' := true; STOPc' := true

**Excerpt: Attacker in the challenge-response client puzzle protocol**

**init**
[] true → Na' := 0; request' := false; h_priority' := false; X' := false; ACK' := false; STOPa' := false; currentS' := 0
**update**
[] ∼ STOPa → request' := true; Na' := Na + 1
[] Pa & (Ns > currentS) & ha & ∼ STOPa → X' := true; currentS' := Ns
[] SYNACK & ∼ STOPa → ACK' := true; STOPa' := true

**Figure 7. MOCHA model of legitimate clients and attackers in the challenge-response protocol of [13].**

**Diff_fromi** Puzzle difficulty level sent by monitor $i$.

**SENDdiffij** Becomes true when monitor $i$ computes the appropriate difficulty level based on the total traffic flow and is ready to send a puzzle to client $j$.

**Ncij** Nonce from client $i$ to monitor $j$.

**reqij** Request from client $i$ to monitor $j$.

**pkts** Array of traffic going to each monitor, *i.e.*, pkts[i] is the number of packets going to monitor $i$.

**difficulty_leveli** Difficulty level of the puzzle client $i$ needs to solve.

**SENDmij_k** Used for communication between monitors. It becomes true when monitor $i$ is ready to send a message to monitor $j$ with information about the level of traffic from client $k$.

**flow_info_toi** Traffic information exchanged between the monitors.

**Model for DoS prevention in the JFKr protocol.** The protocol is described in section 6. An excerpt from our MOCHA model is in fig. 9. In the model, $Ni$ and $Nri$ refer to the initiator's and responder's nonces, respectively; $xi$ and $xri$ refer to their respective Diffie-Hellman exponentials; $ti$ refers to the responder's cookie that the initiator needs to return correctly in the third message of the protocol; $hi$ refers to the hash value computed using key $K_a$.

**Excerpt: Server in the challenge-response client puzzle protocol**

type my_int : (0..2)
private sum : sindexType; allocate_resourceC1 : bool; allocate_resourceA1 : bool; STOPs : bool; currentC1 : my_int; currentA1 : my_int
**update**
[] ∼ STOPs & requestC1 & (NcC1 > currentC1) & ∼ allocatedC1 → PcC1' := true; NsC1' := NcC1 + 1; hcsC1' := true
[] ∼ STOPs & XC1 & ∼ allocatedC1 → SYNACKC1' := true
[] ∼ STOPs & ACKC1 & ∼ allocatedC1 → allocate_resourceC1' := true
[] ∼ STOPs & allocate_resourceC1 & (sum < 2) & ∼ allocatedC1 → allocatedC1' := true; sum' := sum + 1
[] ∼ STOPs & allocate_resourceC1 & (sum = 2) & h_priorityC1 & ∼ allocatedC1 → allocatedC1' := true
[] ∼ STOPs & requestA1 & (NaA1 > currentA1) & ∼ allocatedA1 → PaA1' := true; NsA1' := NaA1 + 1; hasA1' := true
[] ∼ STOPs & XA1 & ∼ allocatedA1 → SYNACKA1' := true
[] ∼ STOPs & ACKA1 & ∼ allocatedA1 → allocate_resourceA1' := true
[] ∼ STOPs & allocate_resourceA1 & (sum < 2) & ∼ allocatedA1 → allocatedA1' := true; sum' := sum + 1
[] ∼ STOPs & allocate_resourceA1 & (sum = 2) & h_priorityA1 & ∼ allocatedA1 → allocatedA1' := true

**Figure 8. MOCHA model of servers in the challenge-response protocol of [13].**

**Excerpt: Initiator in the JFKr protocol**

**init**
[] true → request' := true; Ni' := false; xi' := false; hi' := false; ti' := false; accepti' := false; fail' := false
**update**
[] request & ∼ accepti & ∼ fail & ∼ STOPc → Ni' := true; xi' := true
[] Nri & xri & tr & ∼ accepti & ∼ fail & ∼ STOPc → ti' := true; hi' := true
[] ∼ hri & accepti & ∼ fail & ∼ STOPc → fail' := true; STOPc' := true
[] hri & ∼ accepti & ∼ fail & ∼ STOPc → accepti' := true
[] default → STOPc' := true

**Excerpt: Responder in the JFKr protocol**

**init**
[] true → Nri' := false; xri' := false; tri' := false; hri' := false; accepts' := false; fails' := false
**update**
[] Ni' & xi' & ∼ accepts & ∼ fails & ∼ STOPsi → Nri' := true; xri' := true; tri' := true
[] ∼ fails & (∼ hi' | ∼ ti') → fails' := true
[] hi' & ∼ accepts & ∼ fails & ti' & ∼ STOPsi → hri' := true; accepts' := true
[] default → STOPsi' := true; accepts' := true

**Figure 9. MOCHA model of the JFKr protocol.**

**Excerpt: Filters in the distributed client puzzle protocol**

```
type packets : (0..100)
type no_of_monitors : (0..1)
type traffic : array (0..1) of packets
module Client1
external Diff_from1 : traffic; Diff_from2 : traffic; SENDdiff11 : bool; SENDdiff21 : bool
interface Nc11 : bool; Nc12 : bool; req11 : bool; req12 : bool; pkts1 : traffic; difficulty_level1 : packets
private STOPc : bool
init
    [] true → Nc11' := true; Nc12' := true; req11' := false; req12' := false; STOPc' := false; difficulty_level1' := 0;
pkts1'[0] := 0; pkts1'[1] := 0
update
    [] Nc11 & ∼ STOPc → pkts1'[0] := 1; req11' := true
    [] Nc12 & ∼ STOPc → pkts1'[1] := 4; req12' := true
    [] ∼ STOPc & SENDdiff11 → difficulty_level1' := Diff_from1[0]; STOPc' := true
    [] ∼ STOPc & SENDdiff21 → difficulty_level1' := Diff_from2[0]; STOPc' := true
    [] default → STOPc' := true
endatom
endmodule
module Monitor1
external req11 : bool; SENDm21_1 : bool; pkts1 : traffic; flow_info_to2 : traffic
interface flow_info_to1 : traffic; SENDm12_1 : bool; Diff_from1 : traffic; SENDdiff11 : bool
private STOPs : bool
init
    [] true → STOPs' := false; Diff_from1'[0] := 0
update
    [] ∼ STOPs & req11 → flow_info_to1'[0] := (flow_info_to1[0] + pkts1[0]); SENDm12_1' := true
    [] ∼ STOPs & SENDm21_1 → Diff_from1'[0] := (Diff_from1[0] + flow_info_to1[0] + flow_info_to2[0]); SEND-
diff11' := true; STOPs' := true
endatom
endmodule
module Monitor2
external req12 : bool; SENDm12_1 : bool; pkts1 : traffic; flow_info_to1 : traffic
interface flow_info_to2 : traffic; SENDm21_1 : bool; Diff_from2 : traffic; SENDdiff21 : bool
private STOPs : bool
init
    [] true → STOPs' := false; Diff_from2'[0] := 0
update
    [] ∼ STOPs & req12 → flow_info_to2'[0] := (flow_info_to2[0] + pkts1[1]); SENDm21_1' := true
    [] ∼ STOPs & SENDm12_1 → Diff_from2'[0] := (Diff_from2[0] + flow_info_to1[0] + flow_info_to2[0]); SEND-
diff21' := true; STOPs' := true
endatom
endmodule
```

**Figure 10. MOCHA model of monitors in the new distributed client puzzle protocol.**