Int. J. Inf. Secur.
DOI 10.1007/s10207-007-0042-x

REGULAR CONTRIBUTION

A multi-layer framework for puzzle-based denial-of-service defense

XiaoFeng Wang · Michael K. Reiter

Received: 22 November 2006

Abstract Client puzzles have been advocated as a promising countermeasure to denial-of-service (DoS) attacks in recent years. However, how to operationalize this idea in network protocol stacks still has not been sufficiently studied. In this paper, we describe our research on a multi-layer puzzle-based DoS defense architecture, which embeds puzzle techniques into both end-to-end and IP-layer services. Specifically, our research results in two new puzzle techniques: *puzzle auctions* for end-to-end protection and *congestion puzzles* for IP-layer protection. We present the designs of these approaches and evaluations of their efficacy. We demonstrate that our techniques effectively mitigate DoS threats to IP, TCP and application protocols; maintain full interoperability with legacy systems; and support incremental deployment. We also provide a game theoretic analysis that sheds light on the potential to use client puzzles for incentive engineering: the costs of solving puzzles on an attackers' behalf could motivate computer owners to more aggressively cleanse their computers of malware, in turn hindering the attacker from capturing a large number of computers with which it can launch DoS attacks.

Keywords Denial of service · Client puzzles · Network protocols

X. Wang (✉)
Indiana University at Bloomington, Bloomington, IN, USA
e-mail: xw7@indiana.edu

M. K. Reiter
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: reiter@cmu.edu

1 Introduction

Denial-of-service (DoS) attacks continue to plague today's Internet. Malware and automatic attack tools [18,24,25,35] have substantially lowered the bar for launching massive distributed denial-of-service (DDoS) attacks. Recent attacks have been reported to involve thousands or even tens of thousands of *zombie* computers, which can be acquired from the black market according to Computer Associates.[1] At the same time, resource-intensive services can be susceptible to even low-bandwidth, end-to-end DoS attacks. Besides the well-known SYN-flooding attack, examples of such threats include attacks on authentication services [22] and crafted attacks that drive the data structures used in a service implementation into their worst-case performance [20].

Countermeasures to DoS attacks have been studied for years. Unfortunately, many existing defense techniques are passive in nature: it is the sole responsibility of the defender to detect and filter denials-of-service, while the attacker is spared any penalty for squandering server resources. Such a defense philosophy could be insufficient to defend against vast zombie capabilities to overwhelm the victim. At the same time, it offers little incentive to the owners of Internet hosts to protect their computers from unwittingly joining the zombie fleet—given the negligible interference of the DDoS tools on the compromised machines that host them [18,24,25,35] on the one hand and the significant administrative overhead of malware defense on the other.

Client puzzles [9,22,38] could lead to a potentially more active response to DoS. In this approach, a client solves a computational "puzzle" for requesting service before the server commits resources, thereby imposing a massive computational burden on adversaries bent on generating

[1] http://www.theregister.co.uk/2005/06/03/malware_blitz/.

legitimate service requests to consume substantial server resources. While the idea of client puzzles has been explored to an extent, many important questions remain unaddressed. In particular, it is unclear how to operationalize these puzzles in network protocol stacks to protect the availability of communication resources, while preserving interoperability with legacy systems. In addition, computing puzzles also adds to legitimate clients' load, and in the presence of adversaries with unknown computing power, minimizing legitimate clients' cost is an important challenge.

In this paper, we report on the design and implementation of puzzle techniques to mitigate DoS threats. We envision a multi-layer DoS defense architecture built upon two key puzzle techniques: *puzzle auctions* that offer an end-to-end protection to Internet services and *congestion puzzles* that work in routers to alleviate the threat of bandwidth-exhaustion attacks. Both approaches help control attack flows, are interoperable with legacy systems, and support incremental deployment. We justify these claims through both analysis and empirical evaluations. We also describe a game-theoretic analysis that sheds light on the ability of puzzles to fend off attacks involving a large number of zombies.

2 Related work

2.1 Denial-of-service attacks and countermeasures

An end-to-end DoS attack that has harrassed the Internet for many years is *TCP SYN flooding* [19]. TCP SYN flooding exploits a weakness in the TCP connection establishment protocol (three-way handshake) where a connection state can be *half-open*. A typical TCP three-way handshake proceeds as follows: A client first sends a SYN packet to the server. Upon receipt of the SYN, the server allocates state to hold information associated with the half-open connection and sends back a SYN-ACK packet to the client. The client completes the connection by replying with an ACK packet. In SYN-flooding attacks, attackers initiate many SYN requests without sending ACK packets. This exhausts the server's half-open waiting queue and thus blocks a legitimate client's request from being serviced. Countermeasures to this attack include ingress filtering and SYN-cookies [40]. These approaches are designed to defend against attacks involving spoofed IP addresses and therefore are less effective when adversaries can use (many) zombies' authentic IP source addresses. SYN-cookies also introduce compatibility problems. A server using SYN-cookies does not keep a half-open queue and instead encodes an authentication token into a SYN-ACK packet. However, not all the service parameters can be encoded into the packet, which prevents clients from using certain TCP performance enhancements and transactional TCP [40]. More seriously, if the ACK packet is lost, the server cannot reconstruct the connection state or thus retransmit a SYN-ACK. This does not happen in the original protocol with the SYN-ACK retransmission mechanism [40]. All these weaknesses have been avoided in the approaches we propose in this paper.

Recently, researchers have also discussed other threats to the availability of end-to-end services, which could use resource-intensive operations, such as cryptographic operations, to bring down an Internet server. For example, Meadows [43], Aura et al. [9] and Dean and Stubblefield [22] explored an attack in which a large number of messages with bogus signatures to deplete an authentication server's CPU cycles. Another example is given by Crosby and Wallach [20], who demonstrated that carefully crafted inputs could degrade hash tables to linked lists, and thus force a web proxy to run at its worse-case performance. These attacks can be mitigated by puzzles, including the approaches we propose here.

In a bandwidth-exhaustion attack, attackers generate a large volume of traffic to deplete a network router's bandwidth. Mechanisms to counter this attack include aggregate-based congestion control [36,42,64], overlay systems [4,6,39,45], capability-based approaches [7,33,63], trace-back [5,13,16,21,50,52,53] and filtering [29,37,41,51,55,62]. Aggregate-based congestion control provides mechanisms for detecting and controlling aggregates at a router using an attack signature, and a pushback mechanism to propagate aggregate control requests (and the attack signature) to upstream routers. However, it critically depends on the mechanism by which attacks are detected and an attack signature is formulated, and this can be a source of difficulty against an intelligent adversary that varies its traffic characteristics over time. A website can be protected by a large overlay network, on which individual nodes anonymously route a trusted client's traffic to the website's hidden location. These approaches rely on a CAPTCHA [58] (http://www.captcha.org) to identify human visitors, and therefore may not work for other non-human-driven services such as DNS. Capability-based approaches authorize a legitimate client to establish a privileged communication channel with a server using a secret token (capability). However, they suffer from "denial-of-capability" attack in which adversaries may saturate the link of the server distributing capabilities to prevent clients from obtaining capabilities [8]. Traceback and most filtering methods are designed for countering spoofed traffic, and therefore are of less utility against non-spoofed attack traffic.

2.2 Client puzzles

An inherent weakness of today's Internet applications is that attackers may consume significant server resources at little cost. Client puzzles are a technique that strives to improve this situation: The client is required to commit computing resources before receiving resources. The idea can even help to defend against attacks with large numbers of zombie computers: A study shows that existing DDoS tools are carefully designed not to disrupt the zombie computers, so as to avoid alerting the machine owners of their presence [32]. When client puzzles are used, zombie computers are required to commit computing resources during attacks. This may alert the owner to the attacker's use of this machine and motivate the owner to stop the attack.

To our knowledge, the first proposal for using client puzzles to defend against connection depletion attacks appears in [38]. Client puzzles have also been proposed to similarly protect authentication protocols against denial-of-service [9]. More generally, cryptographic puzzles have been employed for this purpose in the contexts of key agreement [44], defending against junk e-mail [26], creating digital time capsules [48], metering Web site usage [30], lotteries [34,56] and fair exchange [15,31,56]. Recent proposals for puzzle design include puzzle outsourcing [61] and implementation of the dual receiver cryptosystem [23]. Whereas most puzzle proposals impose a number of computational steps to generate a solution, a "memory bound" alternative imposing memory accesses has been devised in an effort to impose similar puzzle solving delay even on clients with very different computational power [3]. Though here we will employ computational puzzles, we expect that memory-bound puzzles could work equally well in our context and will explore this in future work. Gligor presents an attractive approach that utilizes reverse Turing tests as puzzles to prevent automated flooding in network protocols that should be driven by humans [33]. Gligor also offers insightful comments on the weaknesses of computation-based puzzles in providing guaranteed access for end-to-end services during DDoS attacks.

The only prior implementation of client puzzles to protect end-to-end services reported in the literature, to our knowledge, was done in the context of TLS [22]. Although [38] postulates a TCP-based puzzle scheme, it does not report an implementation. In addition, their proposed implementation is incompatible with TCP in that a computer with this puzzle mechanism will not be able to communicate to the computer without the mechanism. The end-to-end proposal we describe here offers better backward compatibility.

To our knowledge, we are the first to propose a puzzle-based defense against DDoS attacks on the IP layer [60]. Feng has argued the importance of implementing puzzles at the IP layer, because otherwise, any upper-level puzzle protection is still vulnerable to the DDoS attacks at this layer [27]. Feng et al. recently proposed an IP-layer puzzle mechanism [28] which, similar to our work [60], uses ICMP source quench messages to transport puzzles. Different from ours, however, their approach does not address issues of collaboration among multiple upstream routers to control a bandwidth-exhaustion attack, and engineering challenges to reduce routers' computation overheads.

3 Puzzle-based multi-layer defense

The objective of puzzle-based DoS defense is to increase the cost for a sustaining attack so as to overwhelm the adversary's attack capability and expose its zombies to intrusion detection systems, while maintaining affordable overheads for legitimate clients and the defender. To this end, the design of a puzzle mechanism must account for the particular type of DoS threat it intends to counter and the constraints imposed by its application domain. As discussed before, DoS attacks can happen to both IP-layer and end-to-end services, each of which raises different technical challenges to development of a puzzle countermeasure. Here we advocate a multi-layer defense which employs separate puzzle mechanisms for IP-layer and end-to-end services; the former addresses the threat of bandwidth exhaustion attacks, and the latter deals with protocol-specific DoS attacks that might induce a sufficiently low rate of traffic so as to be invisible to the IP-layer defenses.

To protect an end-to-end service, we need to embed puzzles into hosts. End hosts usually have sufficient resources to handle verification of puzzle solutions at a high speed. Therefore, the major technical challenge becomes how to minimize legitimate clients' overheads. In addition, end-to-end services are much more complicated than stateless services running on the IP layer, which requires a careful design of puzzles to keep interoperability with legacy systems. On the other hand, an end-to-end service contains built-in mechanisms such as retransmission to accommodate stateful puzzles which could be more effective than a stateless alternative.

An IP-layer defense builds puzzles into routers. A router is a resource-intensive device and could be constrained even in its capability to verify puzzle solutions. More seriously, defeating a bandwidth exhaustion attack may require collaboration among a large number of routers, and the extra burdens of this collaboration could further aggravate individual routers' resource pinch. This suggests that a router-level puzzle mechanism must be very lightweight, incurring a negligible overhead to the defender. A bright side of the IP-layer defense, however, is the simplicity of the stateless IP protocol, which makes interoperability easy to archive.

Our solution for multi-layer DoS defense is composed of two puzzle mechanisms: puzzle auctions which protect end-to-end services, and congestion puzzles which work inside routers. A puzzle auction takes advantage of the stateful end-to-end service to minimize the overhead for a legitimate client to acquire service. We demonstrate an implementation of this technique in the TCP layer that is fully compatible with the original protocol. We have engineered congestion puzzles to minimize the overheads on the defender. We show our approach to be extremely lightweight, incurring negligible overheads to routers. These two techniques are designed for different network devices (end hosts and routers) and can therefore be used both separately and jointly at the user's discretion.

In the following sections, we elaborate the designs of these two techniques and evaluate their efficacy.

4 End-to-end DoS defense: puzzle auction

In this section, we describe puzzle auctions [59], which offers an end-to-end protection of Internet services from DoS attacks. This technique lets each client use the solution of a puzzle it solves to bid for service from an Internet server, and the server allocates its limited resources to the highest bidders—that is, the clients who solve the most difficult puzzles. We design a bidding strategy which allows legitimate clients to bid smartly, raising puzzle difficulty (i.e., their bids) gradually until it is just above attackers' computation capabilities. We also describe an implementation of this technique to the TCP protocol stack to counter the TCP SYN-flooding attack, which demonstrates the efficacy of the technique and its interoperability with legacy systems.

The puzzle auction mechanism can be put to use at the discretion of the Internet server. No ISP or router-level support is needed. This makes the technique easy to deploy. On the other hand, like other end-to-end DoS countermeasures, this approach only works when adversaries cannot deplete an Internet server's bandwidth. In the next section, we describe another puzzle technique which offers an IP-layer DoS protection but requires more resources to deploy.

4.1 The puzzle auction protocol

4.1.1 Model

For our purposes, a puzzle consists of two algorithms: one (possibly randomized) algorithm for generating "candidate solutions" and one deterministic algorithm for verifying whether a candidate solution is an acceptable solution. A *trial* is one sequence of (i) generating one candidate solution using the first algorithm and (ii) verifying it using the second algorithm. We presume that there is no more efficient way to generate an acceptable solution than repeatedly performing trials until the verification algorithm reports a success.

The resource allocation problem we consider can be characterized as a tuple $\langle C, A, S, V, R\rangle$, where: $C = \{c_{i=1,2,\ldots}\}$ is a set of legitimate clients; $A = \{a_{i=1,2,\ldots}\}$ is a set of adversaries; S denotes the server; and $V : C \cup A \rightarrow \mathbb{N}$ is a *valuation function* indicating the maximum number of trials a client (either legitimate or malicious) is willing to perform to obtain the resource in a reasonable waiting period, assuming that it will actually obtain the resource having performed these trials.

The parameter R is a model $R = \langle L, \tau\rangle$ for the server's resources: The server keeps a buffer queue with the length L as resources and allocates a buffer to each request upon deciding to service the request. Every request carries a priority chosen by the client. When the buffer queue is full, the server can deprive a request with low priority of its buffer to make room for a request with higher priority. If a request keeps the buffer for a time interval no less than τ, we say the service (of that request) is complete. Otherwise, we say the service fails. This model is called *buffer model*.

The buffer model is sufficiently general to describe a range of service types with limited resources. To show this, we consider two general categories of resources distinguished by Qie et al. [47]: *renewable resources* such as CPU cycles; and *non-renewable resources* such as processes, ports, TCP connection data structures and locks. First consider services on renewable resources, and let r be the available service rate and T_s be the maximum time the client can wait for the resource. Now consider a client's request that arrives at time T': If among all requests inside the system between T' and $T' + T_s$, the priority of the client's request is within top rT_s requests, then the request will be served before $T' + T_s$. In other words, the adversaries need to submit at least rT_s requests with higher priorities within T_s time units to prevent the request from being served. This situation is described by an instance of our buffer model with $L = rT_s$ and service time $\tau = T_s$. Holding a buffer in this queue for a period no less than τ is equivalent to acquisition of the renewable resource. In services on nonrenewable resources, we treat the length of the buffer queue L as the total resources the server has and τ as the service time. A client must equip its request with a priority high enough to obtain a buffer and keep it until the completion of the service, where the service is preemptive.[2]

In the buffer model, we use an *all-pay auction* to allocate the server's limited resources. In an all-pay auction, all bidders pay their bids before the auctioneer announces the winner. All the payments are forfeited, while only the winner gets the resources. In our mechanism, each client who requests resources is asked to solve a puzzle at a level of difficulty of the client's choosing and attach the solution to its service request. The server maps the difficulty of the puzzle to the priority with a non-decreasing function. When the buffer queue is full, the server uses incoming requests with higher priority to supplant ones with lower priority in the buffer queue. We call this a *puzzle auction*.

[2] Here, we assume nonrenewable resources are preemptive. For example, during the establishment of TCP connections, the server can drop some half-open connections and release the buffers when the half-open queue is full.

4.1.2 The Protocol

In order to detail our auction protocol, we begin by adopting a particular puzzle type. Here we employ a puzzle similar to that of [9], consisting of a "nonce" parameter N_s created by the server and a parameter N_c created by the client. A solution to this puzzle is a string X such that the first m bits of $h(N_s, N_c, X)$ are zeros, where h is a public hash function. We call m the *puzzle difficulty*. We presume that generating candidate values for X is of negligible computational cost, and so treat the verification of a candidate X (i.e., an application of h) as the cost of a trial.

In our mechanism, we take this puzzle construction because it allows a client to select the puzzle difficulty it solves. More specifically, many end-to-end protocols have retransmission mechanisms. We exploit this and the above puzzle formulation to design a bidding strategy for clients to complete the service with minimal computation. Specifically, a client can send its first request without solving any puzzle. If the request is declined, the client knows that the server may be under an attack. Thus, it solves a puzzle and resends a new request with the solution. If the request is declined again, the client further increments the puzzle difficulty in the next retransmission. This process continues until either the client completes the service or her valuation v_c has been reached.

The auction protocol additionally employs the following notation:

- r_c: a service request from client $c \in C$.
- BF: the set of all service requests already in the buffer queue. $|BF| \leq L$.
- DIF: a function mapping each service request to the level of difficulty of the puzzle solution it contains. For a puzzle solution X in r_c, if the initial m bits of $h(N_s, N_c, X)$ are zeros, then $DIF(r_c) = m$. For notational simplicity, we overload the function and denote $DIF(X) = m$.
- CD: the target puzzle difficulty for the client's request.
- $INIT$: the client's initial target puzzle difficulty.
- $INCR$: the value by which the client increments the target puzzle difficulty for its request.
- v_c: maximum number of hash operations client c will perform for this service request.

1. **Client sends service request:**
   (a) Client c sets $CD = 0$ and $X = 0$; and generates a new client parameter N_c.
   (b) Client c does a bruteforce search of the puzzle solution with the difficulty level CD in the interval $X \in [0, v_c]$:

```
While ($DIF(X) < CD$ and $X < v_c$) $X = X + 1$
If ($X = v_c$) exit and report failure.
```

   Client c constructs a request r_c containing N_c and X, and sends the request to S.

2. **Server allocates resources:**
   (a) Server S periodically checks the buffer queue to clear the requests from BF that have completed service.
   (b) On receipt of the client request r_c, server S checks the client parameter N_c in r_c:

```
If (any $r' \in BF$ contains $N_c$) drop $r_c$ and goto 2(c).
```

   Now server S checks BF:

```
If ($|BF| < L$)
        insert $r_c$ into $BF$
else if ($\forall r' \in BF : DIF(r') \geq DIF(r_c)$)
        drop $r_c$ and goto 2(c)
else locate a request $r' \in BF$ with the lowest puzzle
        difficulty among all requests in $BF$, drop $r'$,
        insert $r_c$ and goto 2(c).
```

   (c) Server S sends to client $\hat{c}$ a notification of service failure which contains the current server nonce N_s, where $\hat{c}$ is the client whose request $r_{\hat{c}}$ has been dropped (if any) at 2(b).

3. **Upon receipt of a failure notification, client retransmits:**
   Client c extracts the server nonce N_s from the message, and increases its bid as necessary:

```
If ($CD < INIT$)
        $CD = INIT$
else $CD = CD + INCR$
Goto Step 1(b).
```

In the protocol above, N_c and N_s play roles similar, but not identical, to nonce identifiers as often used in cryptographic protocols. N_s should change periodically, say every T time units, to limit the reuse of puzzles (and their solutions). N_c is constrained merely to not be in use simultaneously by two different requests. Though the adversary can consecutively reuse a puzzle solution X for the same N_c for up to T time units (i.e., before N_s changes), it still needs to generate at least L puzzle solutions of sufficient difficulty in time T to flood the server. In practice, enforcing that only *simultaneous* requests bear different values for N_c, as opposed to all requests, avoids the server needing to store a large list of previously seen nonce identifiers (which could itself pose a DoS opportunity).

The puzzle auction protocol above is efficient in the sense that the client can raise its bid just above the attacker's bids to win an auction. In other words, if the client wins the auction, it wins with the minimum expected computation for the given adversary.

4.1.3 Security analysis

Here, we analyze the security of the proposed puzzle auction mechanism. We consider the following setting: An adversary with Z zombie computers is trying to attack the server S by denying the service over a buffer $R = \langle L, \tau \rangle$. Here, we consider $\tau \ll T$, where T is the duration of the "server nonce period" before N_s is changed, and for simplicity we consider one legitimate client, i.e., $|C| = 1$. Let ξ be the event that the legitimate client $c \in C$ cannot complete the service. The objective of the adversary is to maximize the probability $Pr\{\xi\}$.

For simplicity, we assume that the client c starts bidding at the beginning of a server nonce period, and consequently that the attackers competing with c must, as well. Let $(b_0, b_1, \ldots, b_n)$ be a sequence of bids. The client first bids b_0. If rejected, it continues to bid b_1 and so on. In total, it retries no more than $n + 1$ times. In solving a puzzle, we call a hash operation a *hash step*. We further assume that each zombie and the client can perform hash steps at the same rate s. We call s the *step rate*.

We assume the hash function is a random function (i.e., *random oracle* [11]). That is, for each input, the hash function independently and randomly (with uniform distribution) maps it to an output in the image space. The only restriction is that the same input always yields the same output. In practice, a good candidate for random oracle is MD5 with its output truncated [11]. The random oracle model gives us a geometric random variable for the steps used to solve a puzzle. Specifically, to solve a puzzle with initial m zero bits, a hash step can be viewed as a Bernoulli experiment with a probability of 2^{-m} to succeed. Throughout the rest of the paper, we describe and analyze the puzzle auction mechanism with this model.

Let us first look at the adversary's bidding strategy. Let us assume that the adversary has perfect coordination among zombie computers. Therefore, we can view the attacker as a "super computer" whose computing power is equal to the sum of all zombie computers'. That is, the adversary can perform hash operations at the step rate Zs.

We say a client *set a bid to difficulty level m* if in solving a puzzle, the client bids with the first solution it found whose difficulty level is no less than m. In order for the adversary to cause this bid to be dropped, the adversary must compute L bids of difficulty (at least) m; we are interested in how long it will take the adversary to generate L such bids. Let χ_i^m be the random variable describing the number of steps for computing the i-th bid in all L bids set to difficulty m. $\chi_1^m, \ldots, \chi_L^m$ are i.i.d. random variables. When L is large, we can approximate the total steps for computing the L bids: $\sum_{i=1}^{L} \chi_i^m$ with expectation $E[\chi_i^m]L = 2^m L$. (During this time, the legitimate client can compute roughly $s' \approx \frac{2^m L}{Z}$ steps.)

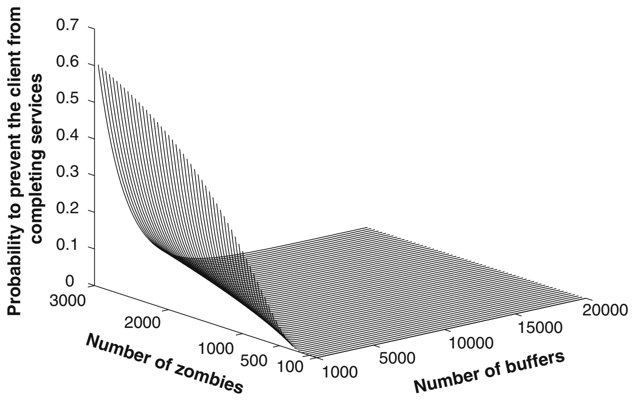


Fig. 1 Security of the puzzle auction

To support the above approximation, we need to investigate the probability that the adversary takes fewer steps to compute the L puzzles. This is answered by Proposition 1.

Proposition 1 *The probability of solving no less than L puzzles with difficulty at least m in $2^{m-1}L$ steps is no more than $exp(-\frac{1}{6}L)$.*

Proposition 1 shows that the attacker's probability to set L bids to difficulty m within $2^{m-1}L$ steps drops exponentially w.r.t. the length of the buffer queue L. For example, if the server has $L = 1024$ buffers, the attacker's chance to overbid is less than e^{-170}.

For simplicity, we ignore the adversary's probability to set all L bids to m within $2^{m-1}L$ steps when buffer queue is sufficiently long. On this basis of this approximation assumption, we estimate the upper bound of the attacker's probability to launch a DoS attack with the following theorem.

Theorem 1 *Under the Approximation Assumption, for legitimate client c with step rate s, service time τ and a bid sequence $(b_0, b_1, \ldots, b_n)$, the probability that the attacker can successfully prevent the client from completing service is*

$$Pr\{\xi\} < (1 - 2^{-b_0})^{\frac{2^{b_0-1}L}{Z} - \tau s} \prod_{i=1}^{n} (1 - 2^{-b_i})^{\frac{(2^{b_i-1} - 2^{b_{i-1}-1})L}{Z}} \tag{1}$$

Figure 1 further illustrates that the probability of attack increases with the number of zombies and decreases with the buffer size.

4.2 Implementation to TCP protocol stack

In this section, we describe our implementation of the puzzle auction mechanism in the TCP protocol stack of the Linux kernel, specifically version 2.4.17. Our implementation effectively defends against connection-depletion attacks on TCP, preserves compatibility with the original protocol and introduces only negligible overheads to the server. Our approach is also interoperable to a degree with clients having unmodified kernels.

To embed our protocol into the TCP protocol stack, the first problem we need to solve is how to determine the client parameter N_c and the server parameter N_s. When establishing a TCP connection, the server decides whether a packet belongs to an existing connection or half-open connection according to its source IP address (SIP), destination IP address (DIP), source port (SP), destination port (DP) and the initial sequence number (ISN). In other words, the server does not allow two connections from the same client for the same ports and the same initial sequence number. Therefore, we can take these parameters, i.e., SIP, DIP, SP, DP and ISN, as the client parameter N_c. This treatment prevents clients from using the same puzzle to make two connections simultaneously. Moreover, it also simplifies the process to verify a puzzle: No extra work is necessary for detecting repeated client parameters because the existing classifier that filters incoming packets automatically does the job.

The server nonce N_s is supposed to change after each nonce period. A straightforward construction is to hash a server secret with a timer which increases for every nonce period. This guarantees that the server nonce changes periodically. Moreover, so as to make an adversary's task more difficult when it cannot eavesdrop on responses to requests bearing a spoofed IP address, we add the client's IP address to the input of the hash function for generating the server nonce. Thus, clients with different IP addresses are given different server nonces. If the adversary sends requests with spoofed IP addresses and cannot intercept the server responses, it will not obtain correct server nonces to compute solutions to puzzles.

To achieve compatibility, we advocate embedding the puzzle auction protocol into the communication flows of the original protocol. In practice, this is feasible because there are numerous *covert channels* in network protocols. In a TCP header, several fields can be used to carry the server nonce and the puzzle solution. During the three-way handshake, the SYN packet from the client has its acknowledgement sequence number empty, into which a 32-bit puzzle solution can be placed. We also take the RST packet as the failure notification and insert the server nonce N_s into its 32-bit sequence number field and, if a larger N_s is desired, in the window size and/or urgent pointer fields (for a total of up to 64 bits).

We roughly describe the TCP puzzle auction as follows: A client first sends a SYN packet without a puzzle solution to the server. After receiving the packet, the server first checks the puzzle difficulty to determine the priority of the request (i.e., the difficulty of the puzzle solved, which should be small, since the client did not intentionally solve a puzzle), and then adds the request to the half-open queue if the buffer queue is not full. Otherwise, the server drops a request with the lowest priority (probably the new packet) and sends back a RST packet with the server nonce generated according to the client's IP address. The receiver of the RST packet uses the server nonce to increase its bid (i.e., compute a puzzle) and retransmits a new SYN with the puzzle solution. If the request is declined again, the client further raises its bid and does a retransmission again. This process continues until the client either receives the SYN-ACK or runs out the maximal number of retransmissions preset by the protocol.

A drawback for client puzzles is that the client needs to install puzzle-solving software. If implemented within the network protocol stack, this may require modifying every client's kernel. Our implementation, however, mitigates this problem: In the TCP puzzle auction protocol, the server determines the puzzle difficulty of a packet by computing $h(N_s, SIP, DIP, SP, DP, ISN, X)$. For a client without a puzzle-solving kernel, the puzzle solution X is fixed in each connection effort. However, it is still able to change the puzzle difficulty with different ISNs. TCP requires that each new session start with a more or less random ISN for preventing TCP hijacking [12]. A client can generate a new ISN by simply starting a new session. Our protocol supports launching new sessions consecutively by using a RST packet to immediately reset the client without a puzzle-solving kernel, thus saving it from doing exponential backoff. By resetting new sessions to query the server with different ISNs, a client will finally hit a puzzle difficulty high enough to complete a connection in most cases. This can be viewed as another strategy to solve puzzles that is undertaken by clients unaware of the puzzle mechanism: Instead of performing hash operations itself, the client treats the server as an oracle to test its solution. We call this strategy *bid and query*.[3]

A problem, however, is that the attackers also can take this strategy. More seriously, they do not need to wait for the answer of the query (SYN-ACK or RST) and thus can continuously generate numerous packets in hopes that many of them can get into the queue. However, this approach is limited by the server's bandwidth. For example, let us look at a 1 Gbps network. Since the shortest TCP/IP packet is 64 bytes, the maximal packet rate of this network is no more than 1,953,125 packets per second (pps). In the Linux TCP implementation, the server drops "old" half-open connections (i.e.,

[3] Semantically, a RST packet usually indicates to the client that no server is listening to a port, thus discouraging the client from reconnecting. To preserve these semantics for a client with a puzzle-solving kernel, a puzzle auction server can signal a dropped bid in some of the unused bits of the corresponding RST header. In the absence of this signal, the client can interpret the RST as meaning there is no server process listening on that port, as usual. A client without a puzzle-solving kernel, on the other hand, will attempt a connection for a preset number of times (without exponential backoff, and so this should proceed quickly) and stop after a number of tries without success.

that have timed out after sending the SYN-ACK at least once) when the queue is full. This may take only 9 s, during which the adversary can submit $1,953,125 \times 9$ packets. Since in expectation, only about $1/2^m$ of these will bear puzzles of difficulty at least m (supposing the adversary is choosing them randomly), if $L > (1,953,125 \times 9)/2^m$ then the adversary will probably fail to consume all L buffers with puzzles of difficulty at least m using this strategy. For example, if $L = 1024$ then a legitimate client will probably be able to succeed with a bid of only $m = 15$, which the legitimate client can generate in roughly 5 s using repeated queries to the server, assuming a round trip time (RTT) of 200 microseconds.

That said, if the attackers can generate such a large throughput, they do not need SYN flooding to attack the server, because they can already exhaust the server bandwidth. This is interesting because it is widely believed that SYN flooding needs a relatively small number of packets and thus is very easy to launch. Our TCP puzzle mechanism, however, raises the bar to this kind of threat and makes it potentially harder to exploit than bandwidth-exhaustion attacks.

4.3 Experimental evaluation

In this section, we report our experimental study of the puzzle auction mechanism in a network environment. Our setup contains three computers: a client, a server and an attacker. The client is an obsolete Pentium Pro 199MHz machine with 64MB memory. The server has an Intel PIII/600 with 256MB memory. Both computers have a 2.4.17 kernel, either a standard one or a customized one with our puzzle auction mechanism. The attacker is strong, having two Intel PIII/1GHz CPUs and 1GB memory. It is also equipped with Linux kernel 2.4.17. Roughly speaking, the attacker has computing power ten times the client's. All these computers are attached to a 100Mbps campus network.

The objectives of this empirical study are: (1) evaluation of the overhead of the puzzle mechanism, (2) test of the performance of the system under SYN flooding attacks.

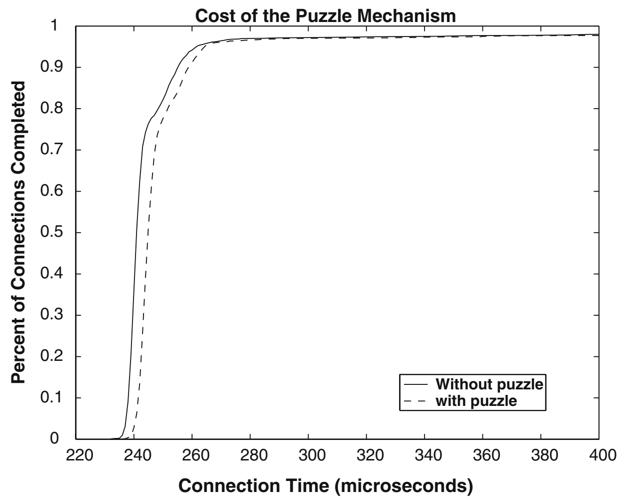
We first study the overhead of the puzzle auction mechanism. When a SYN packet is entering the system, the server first determines its priority. To avoid keeping too much information in the kernel, we configure the server to compute the server nonce for a request on the fly. Therefore, the server needs to take two hash operations to find out the priority of a request: one for the nonce and the other for the puzzle difficulty. The extra costs here are just a little bit above the two hash operations. In our experiment, we made 1,000 consecutive connections each to the server with standard kernel and our customized kernel. The standard kernel gave us the average connection time of 250.8 microseconds. The puzzle auction kernel had an average time of 255.4 microseconds. This empirical evaluation shows that the extra costs brought in by the puzzle auction mechanism are almost negligible: only 4.6 microseconds.

The second experiment is on system performance in the presence of attackers with different computing power. In standard Linux, when a server is under a SYN flood attack, the kernel reduces the number of SYN-ACK retransmissions to two, so as to drop old requests quickly and make room for new requests. That is, a half-open connection may be held about 9 s (3 s for the first timeout and 6 s for the second). In our experiment, we first set the server's retransmission number under attack to 2 and then further reduced it to only one, which took about 3 s to discard a half-open connection after the half-open queue became full. We refer to the first server setting as *Setting 2* (i.e., two retransmissions) and the second as *Setting 1*. The server has a half-open queue with the buffer size of 1,024 and mapped puzzle difficulty levels to priority levels as follows: difficulty levels 0, 1, 2, 3, 4, 5, 6 and 7 to priority level 0; 8, 9, 10 and 11 to 1; 12 and 13 to 2; all difficulty levels k above 13 to $k - 11$.

On the attacker, we installed SYN flooding code capable of generating attack traffic with puzzles of varied difficulty levels. In the experiment, the attacker launched 5 attacks each on the server with different retransmission settings. These attacks set puzzle difficulty to 0, 8, 12, 14, 15. Without solving puzzles, the attacker started SYN flooding at a packet rate of 7,000 pps. This rate easily brought down a server with the standard kernel. With the difficulty level of 8, the attacker was still able to flood the server at the packet rate more than 5,000 pps. However, its capability to generate packets was greatly impaired when the puzzle difficulty went above 12.

In the experiment, we tested two bidding strategies on the client: bid and query (BQ) and incremental bidding (IB). The BQ client had a standard kernel and a small program which made sequential and consecutive connections to the server. The IB client had a puzzle-solving kernel which automatically increased bids via retransmissions. Each experiment lasted until the client completed 500 connections successfully. After each successful connection, the client reset the connection and waited for a period randomly drawn from a uniform distribution between 0 and 150 milliseconds before trying again (so that the next attempt would not immediately reclaim the "opening" that closing the connection created). The *connection time* was measured to the point when the connection succeeds, and was restarted when the following connection attempt was initiated. The *average connection time* was computed by averaging the connection time over the 500 successful attempts.

Our experiment compared the average connection time of these two strategies in various attack and retransmission settings. The results are presented in Fig. 2 bottom, where the X-axis indicates the difficulty level of the puzzles solved by the attacker and the Y-axis is the average connection time for the client. If the attacker did not solve puzzles at all, the



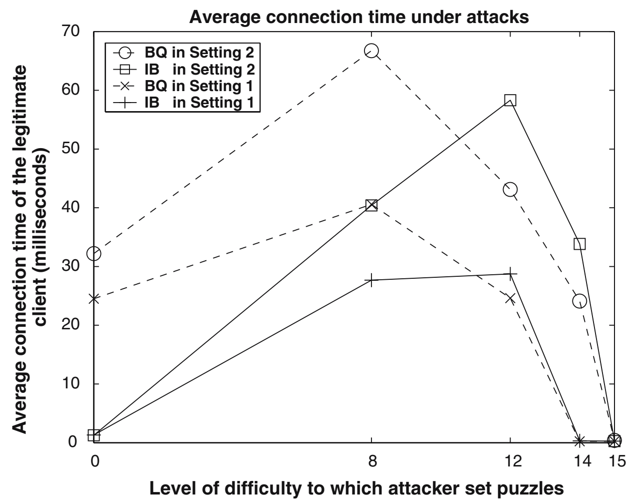


Fig. 2 *Top* Overheads of puzzle mechanism. *Bottom* Average connection time for BQ and IB w.r.t different retransmission settings in the server

client completed connections quickly: In either setting, the average connection time is around 30 milliseconds (ms) for BQ client and 1.3 ms for IB client. With the attacker's bids rising, the connection time was prolonged. For BQ client, the peak came when the attacker set the puzzle difficulty level to 8, 66.737 ms in Setting 2 and 40.581 ms in Setting 1. IB client, however, suffered the maximal delay at difficulty level 12, 58.308 ms in Setting 2 and 28.738 in Setting 1. The expected connection time dropped after the peaks, down to less than 500 microseconds at puzzle difficulty of 15. In general, the client performed well with the puzzle auction server. Even in the presence of strong attacker, a successful connection costs less than 0.1 s on the average.

Although BQ performs comparably to IB, this largely owes to the short RTT in the experiment. Once the server sits outside the client's network, a RTT on the order of milliseconds will raise the connection time. In addition, a BQ client consumes significantly greater server resources. Thus, we emphasize that this should be considered at best a temporary approach to enable a client without a puzzle-solving kernel to participate in puzzle auctions.

5 Router-level DoS defense: congestion puzzles

Puzzle auctions do not suffice when adversaries are capable of generating a large amount of traffic to deplete a victim's bandwidth. In this section, we describe another puzzle technique which works on the IP level and can therefore mitigate the threat of bandwidth exhaustion attacks. This technique, called *congestion puzzles* (CP), is designed for Internet routers to control congestion flows without explicitly detecting attack traffic.

An assumption we made in our research is that adversaries cannot eavesdrop on most legitimate clients' flows. In practice, monitoring a large fraction of legitimate clients' flows is difficult in wide area networks. This assumption allows us to employ very lightweight authentication schemes using sequence numbers or authentication "cookies".

In the follow-up sections, we first survey the CP mechanism, then elaborate on its individual components and finally evaluate this technique theoretically and empirically.

5.1 Congestion puzzles

The puzzle type we adopt for congestion puzzles is identical to what is described in Sect. 4.1.2, which requires a server to periodically update its nonce. We call such a period the *nonce period*.

In order to transmit packets on a congested route, a client should install a *puzzle client* program. This is an application program that interacts with the operating system only through the standard application programming interface (API). This greatly enhances its ease of deployment: e.g., it could be automatically installed from trusted web sites. A client would have incentives to install this program because it increases the client's likelihood to get her packets through during network congestion. In the rest of this paper, we refer to a client with the puzzle client software installed as a "puzzle client".

The CP mechanism is mainly implemented in routers. A puzzle router will trigger the CP mechanism when an outbound link experiences sustained severe congestion, which can be detected by standard methods (e.g., [42]). For instance, a router may monitor the loss rate on the link: If the loss rate exceeds a threshold for several seconds, the router activates the puzzle mechanism.

Once activated, the CP mechanism distributes puzzle parameters (such as a server nonce and difficulty level) to clients, requiring computation flows (puzzle solutions) for traffic traversing the congested link. The manner in which these parameters are sent to the appropriate clients is detailed in Sect. 5.1.1. At a puzzle router's interface, a *puzzle-based rate* *limiter* (PRL) controls the rate of the inbound bit flows on the basis of the computation flows. We describe this mechanism in Sect. 5.1.2.

During a bandwidth-exhaustion attack, a single router usually cannot protect its bandwidth alone. Our solution lets the router push congestion control requests to its upstream routers, which can help prevent the attack flows from converging to the congested router. This is achieved using a *distributed puzzle mechanism* that allows puzzle routers to generate and distribute puzzles and to validate puzzle solutions in a distributed way. We present this mechanism in Sect. 5.1.3.

5.1.1 Puzzle distribution

A congested router needs to propagate a congestion notification and puzzle parameters to the sources (puzzle clients) of the responsible traffic. Moreover, it needs to periodically update its server nonce at these puzzle clients. Here, we present an algorithm that achieves these goals efficiently.

Our algorithm is based on ICMP messages [46]. ICMP is a set of control protocols that provide feedback about problems in Internet communication. An example is PING in which a client sends an echo request to a server to test whether it is reachable; upon receiving the echo, the server replies with the request message. The ICMP header starts with an 8-bit *type* field that determines the rest of the header; so far, 41 of the 255 available type values have been used by various protocols [10]. The PING echo request (ICMP type 8) also has a 16-bit identifier field and a 16-bit sequence number field to aid in matching echos and replies.

Our approach defines two new types of ICMP messages, a *probe packet* and a *puzzle-solution packet*, by which a puzzle client communicates with a congested router. These messages are constructed similar to PING messages, except that they are identified through new type values. A puzzle client uses probe packets to solicit a congestion notification and initial puzzle parameters from a congested router. A puzzle client uses a puzzle solution packet to deliver puzzle solutions to the router. A puzzle client further takes advantage of puzzle solution packets to solicit updated puzzle parameters. So as to permit seamless transition between puzzle parameter updates, routers permit overlapping nonce periods so that both old and new puzzle parameters are allowed for use during a transition period. We denote this transition period by $\mathscr{T}$.

Upon issuing one of these message types, the puzzle client generates and includes a random string called an *authentication cookie* in the message payload. Using this cookie, any router receiving the message can include this cookie in any response to the client, to authenticate itself to the client.[4] In addition to an authentication cookie, a probe message contains a payload of blank space, of length equal to that needed to store puzzle parameters (the difficulty level and server nonce). A puzzle-solution packet contains puzzle parameters, a puzzle solution and blank space for updating puzzle parameters. How routers process these messages is described below.

Monitoring. Each puzzle client monitors network activity of its local system. Whenever the client system visits an IP address, the puzzle client sends probe messages to that address periodically. If there is no congestion, these messages are silently dropped by either the destination host or the router directly connected to that host.

Distributing congestion notifications. Once a puzzle router detects congestion on one of its outbound links, it generates a server nonce, activates a puzzle-based rate limiter (Sect. 5.1.2) and admits a constant flow of probe messages to the congested link from each of its inbound interfaces. For every probe message received, the router inserts the server nonce and puzzle difficulty into its payload (in addition to the authentication cookie), and changes the type of the message to PING echo request. This message will therefore elicit a PING reply to the client containing these parameters.

Updating puzzle parameters

- Upon receiving a PING reply, the puzzle client first checks the reply with the authentication cookie it contains. If correct, the client stops probing and starts sending puzzle solution packets to the IP address it is visiting (Sect. 5.1.2).
- Upon receiving a puzzle solution packet, the congested router utilizes the puzzle in a rate limiting algorithm; see Sec. 5.1.2. If the router has updated its server nonce and/or requested difficulty level, it places these new values into the packet payload (along with the authentication cookie), sets the packet type to PING echo request, and forwards it. This message will thus elicit a PING reply from the destination host to inform the client of the new puzzle parameters.
- If a puzzle client does not receive any PING replies within a period $\mathscr{T}$, it stops sending puzzle solution packets and starts sending probe messages.

The cost of puzzle distribution is modest. The extra traffic caused by probe messages takes only a small portion of bandwidth because a probe packet will typically be much smaller than the packets in a communication flow. To add or update puzzle parameters in a packet, a router only needs to overwrite existing payload fields.

[4] Recall that adversaries are assumed to have only limited capability to eavesdrop and intercept legitimate clients' packets. Other authentication mechanisms, once deployed, also can be used in our approach.

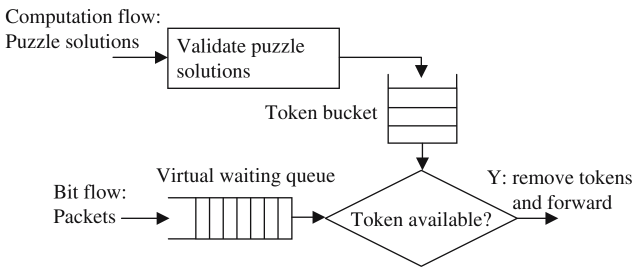


Fig. 3 Puzzle-based rate limiter

5.1.2 *Puzzle-based rate limiter*

During a bandwidth-exhaustion attack, every puzzle client sending packets through a congested link is supposed to generate a virtual "computation flow". The average rate of this computation flow σ_c (average number of hash operations per second) is tied to the rate of the client's bit flow σ_b (bytes per second) through a public *control function* F: $\sigma_b \leq F(\sigma_c, d)$, where d is the difficulty level of puzzles. F is an increasing function of σ_c and a decreasing function of d. Under the random oracle model, we have the average number of hash operations for finding a solution is 2^d. Therefore, a simple construction of the control function is as follows: $F(\sigma_c, d) = \alpha 2^{-d}\sigma_c$, where α is a parameter called *control ratio* measured by *bytes per hash operation*. The control ratio describes the relation between bit flow and computation flow. For example, $\alpha = 10,000$ means that to sustain a bit flow of a rate $\sigma_b = 10,000$ bytes/second, the client is expected to perform 2^d hash operations/second, equivalent to solving at least one puzzle no easier than d per second on the average.

A puzzle router uses the control function to limit the rate of *congestion flows* (flows heading toward the congested link) at its network interfaces. This mechanism is called *puzzle-based rate limiting* (PRL). Without direct observation of computation flow, PRL estimates σ_c as $\sigma_p 2^d$, where σ_p is the rate of puzzle solutions no easier than d. Specifically, PRL implements a *token bucket* and a *virtual waiting queue* at each network interface. For every inbound puzzle-solution packet carrying a correct puzzle solution, PRL adds α tokens to the token bucket at its inbound interface. An inbound packet will be forwarded toward the congested link by removing number of tokens equal to the packet size from the token bucket. When the tokens are depleted, PRL decides on the fate of the packet according to the virtual waiting queue. If there is sufficient room for queuing the packet, PRL forwards it. Otherwise, PRL discards it. We illustrate the mechanism in Fig. 3.

If a puzzle router needs to forward puzzle-solution packets to the next hop (Sect. 5.1.3), these packets also need to be rate limited by the token bucket because they also belong to the congestion flows, even though the puzzle solutions they carry are part of computation flows. This prevents adversaries from using puzzle packets to aggravate the congestion.

The CP mechanism may control the high-bandwidth flows by tuning puzzle difficulty d. PRL has two thresholds: $th_2 > th_1$. If the loss rate of σ_l bits/s at the congested link exceeds th_2, PRL raises d until the loss rate drops just below th_2 but above th_1. If $\sigma_l < th_1$, PRL starts to reduce d. PRL may raise d quickly to suppress attack flows, while lowering d slowly and carefully to prevent intermittent attacks. A problem here is that the puzzle difficulty d only gives a very coarse control of the congestion flows, suppressing σ_b exponentially. This can be complemented by fine-tuning the control rate α to maximize the bandwidth utilization.

The idea of PRL is to constrain the upper bound of σ_b with σ_c and d. For this purpose, *it is not important for a puzzle router to determine whether a particular puzzle solution is correct or not, as long as the router can make a good estimate of* σ_c. Therefore, the router only needs to randomly sample some of the puzzle-solution packets to estimate the ratio of wrong solutions. We will elaborate on this in Sect. 5.2.

The basic PRL mechanism does not differentiate between congestion flows according to their source IPs: As long as they arrive on the same network interface and are destined to the same congestion IP (the destination IP or IP prefix to which a significant fraction of traffic is destined[5]), they are all controlled by the same token bucket. This gives adversaries opportunities to "free ride" on legitimate clients' puzzle solutions, i.e., if their attack traffic arrives on the same interface as the legitimate clients'. This problem would be mitigated with the wide deployment of puzzle routers, since they can better separate the bit flows from different sources based on inbound interfaces. However, when only a few routers have implemented puzzles, free riding could be significant.

Here, we design a simple algorithm to mitigate this problem, called *IP caching*. For each interface, a puzzle router randomly caches a small set of source IPs or IP prefixes from the incoming puzzle-solution packets. For each IP or IP prefix cached, PRL employs a separate token bucket (called *IP bucket*) to control its bit flow. The rest of the congestion flows are handled by a *main bucket* in the same manner as the basic PRL. PRL updates its IP cache with a *least frequently used* (LFU) algorithm: When the cache is full, the IP whose bucket has the fewest tokens added within some period will be merged into the main bucket, to make room for another IP bucket.

5.1.3 *Distributed puzzle verifications*

During a bandwidth-exhaustion attack, a router usually cannot protect itself alone. A cooperative solution, which

[5] This IP adress or IP prefix can be obtained using an approach proposed in [42].

involves upstream routers to help throttle the attack flows, could offer better defense [32]. At a high level, a congested puzzle router may pass a *congestion notification* including congestion IPs and its puzzle parameters to upstream routers, requesting that they activate PRL to prevent attack flows from converging. This, however, may not work well if adversaries manage to send duplicate puzzle solutions through different paths to the victim. Since individual routers do not have a global view, they cannot determine whether a puzzle solution has already been used on another routing path, and thus are unable to prevent the attack flows from reaching the congested router. In this section, we present a *distributed puzzle mechanism* (DPM) to counter this attack.

Our distributed puzzle mechanism requests individual puzzle routers on the puzzle distribution paths to generate their own *path nonces* and attach them to the congestion notification during the puzzle distribution phase. On the path from the congested router to a client, we denote the path nonce of the ith router (starting from the first router upstream of the congested router) by N_i. We call the sequence $N_s|N_1|N_2\cdots|N_{i-1}|N_i$ router i receives from its downstream routers[6] (including itself) the *nonce sequence*.

For two nonce sequences Φ_1 and Φ_2, we denote by $\Phi_2 \in \Phi_1$ if Φ_2 is a prefix of Φ_1; we also denote the part remaining after deleting the contiguous sequence Φ_2 from Φ_1 by $\Phi_1-\Phi_2$. Following we describe the distributed puzzle mechanism through the actions taken by puzzle router i.

Upon receiving a congestion notification M on interface I

1. Randomly generate a path nonce N_i, append it to M and save the nonce sequence $\Phi_i = N_s|N_1|\cdots|N_i$ and the congestion IPs.
2. Forward M to the upstream neighbors from which packets with congestion IPs come.
3. Activate PRL on all inbound interfaces except I to control the flow with congestion IPs.

Upon receiving a probe packet, process as a normal probe packet, using Φ_i as the server nonce.

Upon receiving a puzzle-solution packet with a nonce sequence $\Phi = N_s|N_1|\cdots|N_k|N_c$

1. if ($\Phi_i \notin \Phi$) then drop the packet and return.
2. if ($\Phi - \Phi_i$ appeared before) then drop the packet and return.
3. Validate the puzzle solution (Sect. 5.2.1) and then save $\Phi - \Phi_i$ for checking repeated puzzles.
4. Forward the puzzle-solution packet to the next hop.

For solving puzzles or validating solutions, nonce sequences are treated as server nonces. Each router i also takes the sequence of path nonces starting from its upstream neighbor to the puzzle client (i.e., $\Phi-\Phi_i$) as the client nonce, which we call the *client nonce sequence*.

By using path nonces, DPM gives different responders different puzzles (nonce sequences), thus preventing the adversary from replaying the solutions via different paths.

[6] Recall N_s is the nonce of the congested router.

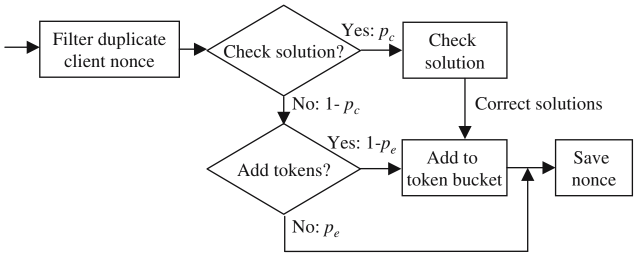


Fig. 4 Probabilistic Validation. “A:p” refers to an event A (Yes or No) that happens with a probability p

5.2 Implementation costs

In this section, we show that with a proper management, the overheads of the CP mechanism (in terms of both computation and memory) can be easily afforded by a modern router.

5.2.1 Probabilistic validation of puzzle solutions

Essentially, our puzzle-based rate limiting controls the rate of bit flow σ_b according to the rate of computation flow σ_c. This implies that a puzzle router does not need to know whether a particular puzzle solution is correct, as long as it can reasonably estimate σ_c. Probabilistic validation (PV) is based on this idea. Specifically, a puzzle router employs a *sampling probability* p_c to determine whether to validate an inbound puzzle-solution packet; if the packet goes without being validated, the router tosses another coin biased to a *false probability* p_e (see below) to decide whether to add tokens into the token bucket or not. This process is illustrated in Fig. 4.

The false probability p_e represents the ratio of false puzzle solutions contained in the current computation flow, which is estimated from puzzle solutions sampled in the recent past. This raises two research questions, however: (1) how to choose the sampling probability p_c and (2) how to estimate the false probability p_e.

Intuitively, one can use a constant p_c, that is, sample every puzzle-solution packet with the same probability. This treatment, however, does not work well due to the variation in the arrival rate of these packets that the router must accommodate. In particular, an adversary may produce a large volume of such packets in an effort to depleting a puzzle router's CPU resources. Therefore, we employ a *dynamic sampling probability* such that when the arrival rate is within a puzzle router's processing capability, most puzzle solutions will be validated. When the arrival rate grows, the router reduces the number of samples to protect its CPU resources.

We design a very simple dynamic sampling method. At time t, a puzzle router first estimates the packet arrival rate of puzzle-solution packets σ_a^t with a typical exponential-averaging rate estimator [54]. Then, the router compares σ_a^t with a *sampling index* η, which roughly indicates the average number of hash (e.g., MD5) computations the router is willing to perform in one second for every interface, to compute the sample probability at time t as $p_c^t = \min\{\frac{\eta}{\sigma_a^t}, 1\}$. This sampling probability changes dynamically with the packet arrival rate of puzzle-solution packets.

A follow-up question is how to estimate the false probability p_e. Since every sample has been chosen with a different probability, a simple averaging over all the validation results gives a biased estimate of the ratio of false puzzle solutions. Here, we present two simple estimators which works well with dynamic sampling: *weighted averaging* (WA) and *exponential averaging* (EA).

At time t, WA averages the validation outcomes of the sampled puzzle solutions, weighted by the inverse of the sample distribution over all puzzle-solution packets received before t. In other words, it gives the samples drawn with small p_c heavy weights and these with large p_c light weights.[7] Specifically, WA works as follows. The router keeps the total number of puzzle-solution packets received before t: Θ_t and the sum of all the sampling probabilities before t: $W_t = \sum_{t' \leq t} p_c^{t'}$. On validating a puzzle solution at time t, the router increases the total number of samples: $n \leftarrow n+1$ and updates a value V. If the puzzle solution is correct, $V \leftarrow (1 - \frac{1}{n})V$; otherwise, $V \leftarrow (1 - \frac{1}{n})V + \frac{1}{np_c^t}$. Then the estimate of the false probability p_e^t can be computed as: $p_e^t = \min\{\frac{W_t V}{\Theta_t}, 1\}$. The router can reset all the parameters whenever the congested router changes the puzzle difficulty.

Sometimes, adversaries may change their strategy during a DDoS attack. For example, they could honestly solve puzzles initially, and then suddenly produce large numbers of false solutions. In this case, an estimator that can quickly adapt to the adversary's behavior is desired. One such estimator that works well in practice is exponential averaging. EA is as simple as follows: If the router samples a correct puzzle solution at time t, then $p_e \leftarrow (1 - \lambda)p_e$; otherwise, $p_e \leftarrow (1 - \lambda)p_e + \lambda$, where $0 < \lambda < 1$ is a small constant. The idea of EA is to bias the false probability towards the most recent observations. Therefore, it reflects the adversaries' recent strategy. It does not even need to compensate for the dynamic sampling, given that an appropriate λ is chosen to give a weight to the new sample. In our experiments, we have observed that EA achieved a slightly better performance than WA.

Both EA and WA make a good estimate of σ_c with a very small number of samples. Our experiments show that during a bandwidth-exhaustion attack, a router sampling no more than 80 puzzles per second (80 MD5 operations[8]/second) controlled congestion flows effectively. Such computing loads would effect a modern router negligibly. For example, a route-switch processor (RSP) of Cisco 7500 series router [2] has a MIPS 4600 CPU with a clock speed ranging from 100 to 250 MHz. In his paper on MD5 performance [57], Touch shows the performance of optimized MD5 on a comparable CPU MIPS 4400 (with a clock speed 150 Mhz) can achieve a rate of about 51.2 Mbps. A puzzle-solution packet usually does not exceed 100 bytes. Therefore, performing 100 MD5 operations per second takes only about 0.16% of the router's CPU time.

5.2.2 *Minimizing the memory for storing client nonces*

In order to prevent adversaries from reusing puzzle solutions, a puzzle router is expected to keep all client nonce sequences (except those in invalid puzzle solutions) throughout a nonce period. This may constitute a considerable memory expense. In this section, we show how to use a space-efficient data structure called *Bloom filter* [14] to compress the required storage to a size acceptable to a modern router.

A Bloom filter is implemented using a large bit vector with m bits. The bit vector initialized to zeros. For every new puzzle-solution packet, the Bloom filter employs k independent uniform hash functions to map the client nonce sequence to k bits in the vector and then sets each of these bits to 1. Bits can be set multiple times.

A duplicate client nonce sequence can be easily detected by computing the k bits with k hash functions. If any one of these bits is zero, the client nonce sequence has not appeared before within the current nonce period. If all bits have been set, it is highly likely that the puzzle solution is duplicate. It is possible that some unused nonce happens to collide with these stored in the Bloom filter, thereby causing a *false positive*. However, the probability of the false positives can be controlled.

The hash functions implemented in the Bloom filter can be very light-weight, e.g., much more efficient than MD5, since no cryptographic strength is required for these hash functions. Specifically, it does not have to be difficult to find the preimage given a hash image. Previous research presents

[7] Essentially, WA and dynamic sampling are similar to the *importance sampling* in statistics, which concentrates sampling on the important part of a dataset. The difference is that in a DDoS attack, it is hard to tell which part of a computation flow is important: Adversaries with perfect coordination among their zombies can manipulate the flow. Here, the dynamic sampling just serves for protecting routers from exhausting its computing resources.

[8] MD5 operation here refers to the operation of computing an MD5 hash function with the puzzle parameters as input.

promising candidates, e.g., the salted CRC-32 [52], which can perform at link speed.

One prominent property of a Bloom filter is that there is an explicit tradeoff between the size of the filter and the probability of false positive. Let n be the maximal number of nonces a puzzle router plans to store. After the Bloom filter is full, the probability of a false positive is: $P = (1-(1-\frac{1}{m})^{kn})^k \approx (1-e^{\frac{-kn}{m}})^k$. For example, with $m = 16n$ and $k = 8$, the false positive probability is about 0.00058. This gives legitimate clients 25 hops away a mistaken reject rate less than 0.015.

Modern routers could afford the memory for implementing a Bloom filter. Snoeren et al. even suggest to use this method to record the trace of every packet traversing a core router [52]. Their research further shows that mere software support is sufficient for slow-to-medium speed routers (up to OC-12). With proper hardware support, it works for fast routers (OC-48 and faster) [49]. Our approach only records the trace of nonces in a nonce period and thus requires smaller memory in general.

For example, a Cisco 7500 series router has a packet switching capability of up to 2.2 M packets/second.[9] Given a (m/n) ratio of 16, if all these packets are puzzle solutions, a puzzle router needs 88 MB memory to keep all the client nonce sequences within a nonce period of 20 seconds. On the other hand, even the memory on a single RSP can be extended to 256 MB or more [2]. Actually, this throughput of puzzle-solution packets is unreasonable because these packets are used to reserve the bandwidth. Therefore, a puzzle router can use some standard rate limiter [42] to limit the arrival rate of puzzle-solution packets to an appropriate ratio of its switch/forward capability, before the puzzle flow is processed by PRL. Here we take a ratio of $\frac{\kappa}{\alpha}$, where $0 < \kappa \leq \alpha$ is a constant. With $\kappa = 1$ $\alpha = 10,000$, the size of the Bloom filter is reduced to 1.1MB, which can be easily built into modern routers.

5.3 Evaluation

In this section, we evaluate the performance of congestion puzzles under bandwidth-exhaustion attacks. Our experiment is based on NS-2 [1], the most widely used network simulator, and CAIDA's Skitter map [17], a traceroute map of real Internet topologies. Due to the limitations of NS-2, we had to keep the scale of our simulation within thousands of nodes. However, we also limited the bandwidth of congested link to only 20Mbps. We believe that a realistic network with higher bandwidth (e.g., 1Gbps) could withstand larger scale attacks by using our techniques.

[9] The length of the packet is usually set to 1000 bits.

From the skitter map, we randomly selected 1,500 paths. Each path ends with an end host. We randomly chose 500 hosts as legitimate clients. The number of zombies was set to 100, 300, 500, 800 and 1,000. Their locations were also randomly drawn from the end hosts.

On the basis of the 1,500 paths, we constructed a network with NS-2. A congested link which was the adversaries' target connected a web server to the network. The bandwidth of the congested link was set to 20 Mbps and every other link to 30 Mbps. Each legitimate client simulated traffic for browsing web pages with the NS-2 web traffic generator. Each adversary produced UDP packets at a constant rate of 300 Kbps to target the congested link. The minimum rate of the attack traffic (with 100 adversaries) was 30 Mbps and the maximum rate (with 1,000 adversaries) was 300 Mbps.

The congested router has a nonce period of 20 s. Each end host installed a puzzle-client agent. On receiving congestion notification, each puzzle client started to continuously solve puzzles of the difficulty level d given by the congested router. We set the time to perform one MD5 operation to 10 microseconds. Each puzzle client determined the number of MD5 steps n for finding a puzzle solution according to a geometric random variable with a distribution $(1-2^{-d})^{n-1}2^{-d}$. This realistically simulated the puzzle-solving delay. After solving a puzzle, the puzzle client sent a puzzle-solution packet to the congested router.

5.3.1 *Puzzle difficulty*

We first evaluated the performance of congestion puzzles using different levels of puzzle difficulty. Figure 5 top depicts the impact of puzzle difficulty (x-axis) on the legitimate clients' packet acceptance rate (the number of packets sent vs. the number of packets received by the web server). Here, difficulty level 0 represents the case without congestion puzzles. We note that the sending rates of both attackers and legitimate clients are unaffected by the puzzle solving difficulty, as puzzle solving (by puzzle clients) is decoupled from application traffic, though obviously difficulty impacts this traffic reaching the target.

Without puzzles, legitimate clients stood little chance to connect to the web server. The situation improved with increase of the puzzle difficulty. When there were less than 300 zombies, the peak of the acceptance rate arrived with $d = 18$, more than 90%. In the presence of more zombies, more difficult puzzles were expected for choking the attack flows. Especially, in the case that the number of zombies exceeded that of the legitimate clients, we needed $d = 20$ to secure an acceptance rate above 85%. Higher difficulty levels were unnecessary and adversely affected legitimate clients' packet acceptance rates.

Adversaries' traffic was substantially controlled with the increase of puzzle difficulty. This is presented in Fig. 5

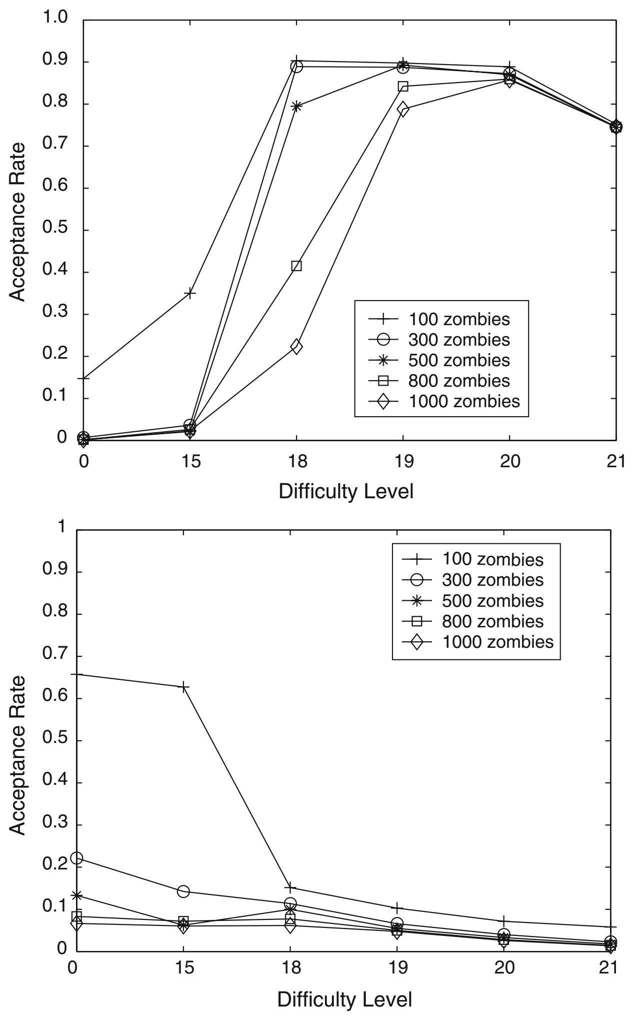


Fig. 5 Impact of puzzle difficulty on the packet acceptance rate: *top* Legitimate clients, *bottom* Adversaries

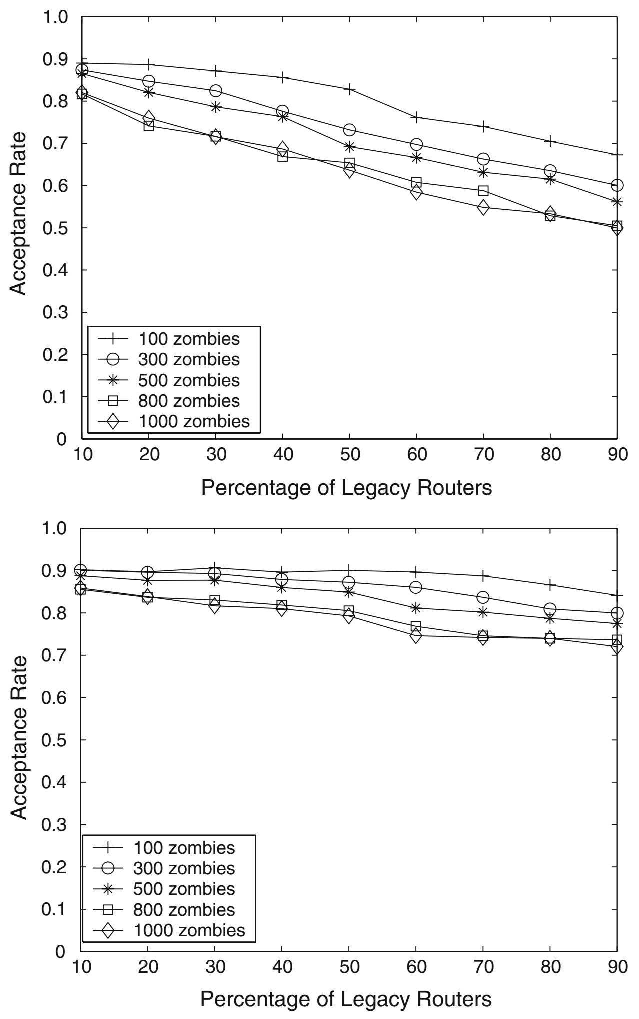


Fig. 6 Legitimate client acceptance rate for partial distribution: *top* without IP caching, *bottom* with IP caching

bottom. This experiment suggests that by tuning puzzle difficulty appropriately, congestion puzzles can effectively contain a bandwidth-exhaustion attack.

5.3.2 Partial deployment

In this experiment, we investigated the performance of congestion puzzles when puzzle routers were only partially deployed. In these experiments, we randomly chose some percentage of routers out of the network as legacy (non-puzzle) routers. However, we fixed the routers close to the congested router (within five hops) to be puzzle routers; deployment was "partial" only further away. We believe this setup will reflect a situation that might occur in practice were congestion puzzles adopted, in which puzzle routers are deployed in clusters to defend important stub networks. Note that these routers are included in the calculation of the percentage of legacy routers. In this context, we tested congestion puzzles with and without IP caching, which helps achieve a fine-grained control of the inbound flows when adversaries attempt to free ride on legitimate clients' computation flows.

We present the experimental results in Fig. 6, in which the x-axis represents the percentage of legacy routers out of all the routers in the network, and the y-axis is the acceptance rate of legitimate clients' packets. In the cases that the number of zombies did not exceed that of legitimate clients, we set the puzzle difficulty $d = 19$, otherwise, we set $d = 20$.

The figure on the top describes the experiment without IP caching. Legitimate clients' acceptance rate decreased with the increase of the percentage of legacy routers. Until the legacy routers took 50% of the whole network, the acceptance rate kept above 60% even with 1,000 zombies. The mechanism also performed well with a small number of zombies. For example, with 90% legacy routers and 100 zombies, near 70% acceptance rate was achieved. However, a minimal deployment (90% legacy routers) plus a large number of zombies (1,000) intensified the free riding problem, thereby reducing the acceptance rate to about half.

The free-riding problem could be effectively suppressed with IP caching. In the figure on the bottom, we show the results of the experiment in which each puzzle router randomly cached 10 IPs/port. This treatment made the mechanism perform well even when only a very small fraction of routers supported puzzles: With 1,000 zombies and 90% legacy routers, more than 70% of legitimate clients' packets were still able to reach the web server in spite of the bandwidth-exhaustion attack.

5.3.3 Performance of probabilistic validation

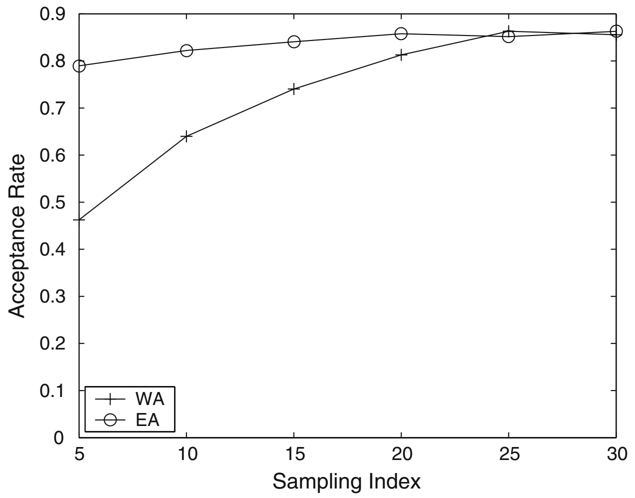
In this experiment, we empirically studied the idea of probabilistic validation when zombies generated false puzzle "solutions".

We considered the adversaries with two strategies. With a stationary strategy, each zombie decided on whether to generate a false puzzle solution according to a fixed probability p drawn uniformly at random. With a dynamic strategy, a zombie randomly chose two probabilities $p_1 < p_2$ and a switching time $0 < t < 20$, and generated false puzzle solutions with p_1 before t and then switched to p_2.

We evaluated PV with dynamic sampling and either the weighted averaging (WA) or exponential averaging (EA). To protect routers from spending computation on a flow containing hardly any correct puzzle solutions, we set a policy that once a router made more than 300 samples from a puzzle solution flow and found its false probability always above 0.95, the router would stop checking the flow and drop all the packets.

In Fig. 7, we present the experimental results. In the experiment, we fixed the number of zombies to 1,000 and set the puzzle difficulty to 20. The x-axis in the figures gives the sampling index, a rough indication of the maximal number of MD5 operations each router was willing to perform per network interface every second. The top figure shows the case with stationary adversaries. WA was pretty sensitive to the sampling index. It performed well after the index exceeded 20. In contrast, EA behaved in a more stable fashion, only varying a little (about 5%) while the index increasing from 5 to 30. Both estimators helped the CP mechanism achieve more than 85% acceptance rate with large index.

Surprisingly, adversaries gained nothing from the dynamic strategy. Actually, both WA and EA performed better there. Two factors might have contributed to this result. First, both estimators (especially EA) might be quite capable of catching up to the adversaries' strategy. Second, since we measured the acceptance rate over the packets transmitted in



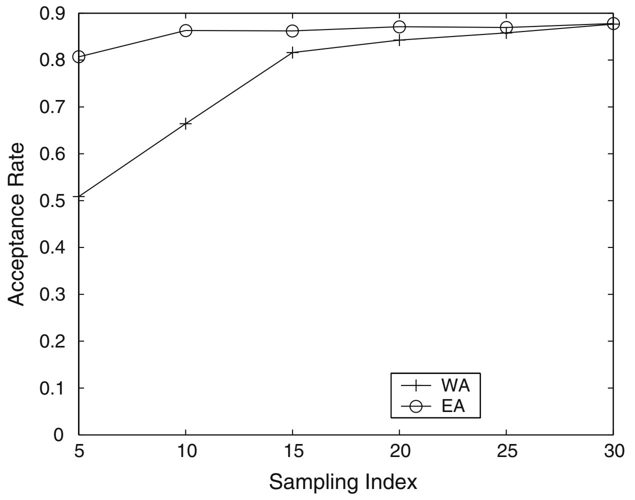


Fig. 7 Legitimate client acceptance rate for probabilistic validation: *top* stationary strategy, *bottom* dynamic strategy

the whole nonce period, the adversaries' relatively honest behavior (before switching time t) might help to improve the final result.

In both experiments, routers made few samples. The maximal MD5 rate was lower than 80 per second for the most heavily-loaded router; the average rate was lower than 10 per second. Such computation load is very affordable for a modern router.

6 Incentive engineering

Intuitively, it seems that an adversary only needs to obtain more zombies to defeat puzzles. This, however, comes with costs. Here, we present a very simple game-theoretic analysis which shows that the puzzle technique can actually engineer Internet users' incentives and make a large-scale attack more difficult to happen.

The game of DDoS attack is not just played by the adversary (maybe irrational) and the victim. Between them, there stand Internet users who are rational, acting on their own interests. The adversary attempts to steal these users' resources to attack the victim, while keeping the users' cost of unwittingly participating in the attack much lower than the cost of removing the malware [32]. Previous research shows that such a strategy, taken by most DDoS tools, minimizes the motivation that the computer owners have to protect their computers [32]. This however could be changed with the introduction of puzzle techniques.

Let $\nu(\sigma_c, d)$ be a user's cost of computing puzzles with rate σ_c and difficulty level d. We consider the case that the number of zombies is large enough to launch a DDoS attack even when individuals act exactly as legitimate clients. In this case, both the zombie and the client incur the same computing costs $\nu(\sigma_c, d)$. Let $\hat{\nu}$ be a constant cost for stopping the attack (removal of malware) or taking measures to prevent malware infection. $\hat{\nu}$ could be the expected expense for buying anti-virus software, or just effort to download and install patches. Let υ be a legitimate user's valuation of the service the victim provides.

The owner of a zombie computer gets no profit from attacking the victim. Therefore, her objective is to minimize the cost. If $\nu(\sigma_c, d) < \hat{\nu}$, she lets the attack continue. Otherwise, her optimal strategy is to cleanse her computer of the malware. This also applies to those whose systems are vulnerable to malware exploits and thus easy to acquire by the adversary. If $\nu(\sigma_c, d) < \upsilon$ and the DDoS attack stops, a legitimate client enjoys a profit of $u = \upsilon - \nu(\sigma_c, d)$ which positively relates to the profit of the target of the attack (the victim). Otherwise, the client cannot obtain or afford the service and thus neither the client nor the victim profits. This implies that their common interests is to maximize u. For simplicity, we treat them as a single player throughout the following analysis.

We model the interaction between owners of zombies and potential zombies (the legitimate systems which are weakly protected and thus vulnerable to malware exploits) and the victim as a *sequential game* in which the victim first declares puzzle parameters which determines $\nu(\sigma_c, d)$ and then the owners choose their strategies (attack or stop attack). A rational owner will choose to stay out of the attack once the cost of attack exceeds the cost of stopping it, formally $\nu(\sigma_c, d) \geq \hat{\nu}$. When this happens, the victim's profit is determined by the valuation of the service and the cost of solving puzzles which equals to $\hat{\nu}$, the cost of stopping the attack. Such a profit remains positive if $\upsilon > \hat{\nu}$. If this condition holds, both victim and the owners' rational moves will be locked into a fixed point: The victim's optimal strategy is to set $\nu(\sigma_c, d) = \hat{\nu}$ to maximize the profit u, and zombies owners' optimal strategy is to stop the attack to minimize the loss. These strategies form a *Nash equilibrium* for the sequential game in which no player can be better off by taking another strategy given the other sticks to its strategy. In practice, a problem for the analysis is that the owners of the zombies might not be aware of the presence of malware on their systems. This can be solved by introducing an intrusion detector to monitor the anomalous usage of CPU resources, which suggests potential malware activities.

The above analysis indicates that puzzles mitigate the threat of large-scale DDoS attacks if $\upsilon > \hat{\nu}$, that is, the legitimate clients' valuation of service must exceed the costs for prevention or removal of malware from an infected system. We believe this is achievable: For example, customers' benefits of having maintenance service from Microsoft are usually much higher than the little inconvenience resulting from downloading and installing the most up-to-date patches. By setting puzzle parameters properly, a puzzle mechanism encourages the owners of zombies to stop contributing to an attack, which makes it harder for an adversary to sustain a large-scale DDoS attack. Remember that we consider a very adverse situation where even if individual zombies behave just like legitimate users, the aggregate of their traffic can still overwhelm the victim. When this happens, most other countermeasures no longer work[10] because zombies' and legitimate clients' traffic becomes indistinguishable.

7 Discussion

Bidding information during a puzzle auction is exchanged through some unused bits in packets' TCP headers, which may cause these packets to be processed outside a router's fast path. However, this happens only when the server is about to run out of its half-open connection resources, which is very unusual in the absence of attacks. Even in this case, only the SYN packets and RST packets for unsuccessful connection attempts and the SYN packet for a successful connection[11] go through the router's slow path, because other TCP packets in an already established connection are kept unchanged in our approach and thus still go through the fast path. Actually, the fast path only works on packets which have previously been sent to the same address, and therefore the first packet of a connection (the SYN packet) usually has to be switched by the slow path. This suggests that the impact of our approach on the normal communication between the client and the server could be very small.

If a server's half-open queue is too short, even some common events such as a background port scan could cause the

[10] A Turing test may still work. However, so far it only works for protecting web servers. Even in this case, it prevents many benign software agents such as crawlers to collect data for their owners.

[11] The packet carries a puzzle solution in its acknowledgement sequence number field.

needed bid to increase. However, this has more to do with the misconfiguration of the server than the weakness of our approach. Puzzle auctions actually mitigate this problem by offering legitimate clients a way to connect to the server if they are willing to pay the computation costs.

The puzzle techniques we propose here work on TCP and IP layers, and therefore can protect all kinds of services above them. However, the overheads of these techniques could be perceived differently in different application protocols. For example, SSH (port 22) itself involves intensive computations for encrypting and decrypting data, which could make the delay caused by puzzle solving less obvious than that in other protocols such as HTTP.

Puzzle auctions can mitigate the threats of application-level resource depletion attacks. A prominent example for such attacks is the attack on computationally-intensive services, such as public-key authentication. Previous research shows that puzzles work effectively against this attack in principle [22]. Our approach can be used to set the right level of puzzle difficulty to defeat the attackers while maintaining reasonably small computation overheads for legitimate clients.

Puzzle auctions could also contribute to the mitigation of bandwidth-exhaustion attacks when they are used along with other techniques. Both capability-based and overlay-based systems require a client to convince a server (either a capability issuer or an overlay node) of its legitimacy so as to acquire access to a protected communication channel. Although a CAPTCHA can serve for this purpose to some extent, it is not general enough for many services not driven by humans. Our puzzle auction approach can be applied to these systems to control adversaries' capability to attack through protected channels. Indeed, it has been reported that some proposals, e.g., SIFF [63], have already implemented our technique.

Congestion puzzles mitigate the congestion on a router's output link by controlling the traffic flowing toward that link. This does not affect the servers receiving the packets from other links of the router. A problem here is that our approach could be a bit greedy: it always allocates the bandwidth according to the computation flows clients generate. This may starve those with limited computation power. A potential mitigation is to first grant everyone a fair quota, and then use computation flows to allocate the remaining bandwidth.

Combined with other DoS countermeasures, congestion puzzles could mitigate bandwidth-exhaustion attacks even with a very limited deployment. For example, we might use a CP-protected edge router to help distribute capabilities for a capability-based system, which prevents adversaries from capturing a large number of capability tokens. In addition, our CP mechanism could work together with some network intrusion detectors (NID) to price the flows carrying attack signatures with a higher puzzle difficulty level, which limits the throughputs of these flows. This treatment differs from other signature-based filtering in that a suspicious flow will only be "taxed" rather than cut off, which reduces collateral damage to legitimate clients' traffic caused by the false positive of the NID signatures. Another way to make the CP technique more effective is using referral and reputation mechanisms. A puzzle router could give easier puzzles to the flows from some reliable sources, for example, a friendly ISP's domain, which helps further reduce legitimate clients' overheads. To prevent IP spoofing in this case, a shared secret key could be issued to these trusted domains to generate puzzle parameters.

Both of our puzzle mechanisms require clients to install software. However, such software could be made very lightweight and easy to deploy. For example, a script will be enough to serve as a puzzle client in most cases. Although we implemented TCP puzzle auction to the Linux kernel, a non-kernel alternative which works as a simple firewall or proxy on the client is also plausible.

A limitation of our approaches is the use of computation-based puzzles which have been demonstrated to be less socialistic—incurring substantial overheads to those with weak computation power, such as PDAs. However, the general architectures of both puzzle auction and congestion puzzles could accommodate other types of puzzles, for example, the memory-bound puzzles which are more fair to clients with different computation resources.

8 Conclusions and future work

Denial of service attacks pose a grave threat to the current Internet. This threat seems unlikely to fade away in absence of an active defense technique which pressures attackers' resources and raises the costs for delivering attack traffic. Client puzzles offer such a defense: An adversary cannot seize the victim's resources without committing its own resources first, which therefore limits its attack capability. In this paper, we describe our research on a multi-layer DoS defense architecture built upon two key puzzle techniques working on different network layers: puzzle auctions, which provide end-to-end protection, and congestion puzzles, which control attack flows on the IP layer. We elaborated on the designs of these two approaches, and reported our evaluations of their efficacy and usability. These approaches have been demonstrated to be effective in mitigating DoS threats to the IP, TCP and application layers; to be interoperable with legacy systems; and to support incremental deployment. We also analyzed the puzzle technique using game theory, shedding light on the potential of employing this technique to motivate Internet clients to aggressively protect their computing systems, making it more difficult for an adversary to capture a large number of zombie computers.

Appendix

A: Selected proofs

Proof for Proposition 1 Our proof requires the following Hoeffding Bound:

Lemma 1 Hoeffding Bound *Let $X_1, X_2, \cdots, X_n$ be a set of independent, identically distributed random variables in [0,1], and let $X = \sum_i X_i$. Then:* $\Pr\{X - E[X] \geq \epsilon E[X]\} \leq \exp\left(-\frac{1}{3}\epsilon^2 E[X]\right)$.

Under the random oracle model, we can take steps for solving puzzles as a sequence of i.i.d. Bernoulli random variables, with probability 2^{-m} for the outcome 1, that is, found the solution to the puzzle; and probability $1 - 2^{-m}$ for 0. Let $\chi_1, \chi_2, \ldots$ be this random variable sequence. Therefore, we have that in $2^{m-1}L$ steps, the total number of puzzles being solved is $\chi = \sum_{i=1}^{2^{m-1}L} \chi_i$. With the linearity of expectation, it is easy to get $E[\chi] = L/2$. According to Lemma 1, we conclude: $Pr\{\chi - E[\chi] \geq E[\chi]\} \leq \exp\left(-\frac{1}{3}E[\chi]\right) = \exp\left(-\frac{1}{6}L\right)$. □

Proof for Theorem 1 To prevent the client from completing the service, the attacker must block all of c's $n+1$ bids. With the Approximation Assumption, we ignore the probability that the adversary generates L bids with difficulty level at least b_i within $2^{b_i-1}L$ steps. The total Z zombies allow the adversary to complete hash steps Z times faster than the client c. That means, if the client c can set its bid to difficulty b_i within $2^{b_i-1}L/Z - \tau s$ steps, the adversary cannot prevent the client to complete the service. On the other hand, from the adversary's viewpoint, this is the necessary condition to block the client. Let ξ_i be the event that the adversary successfully prevents c from getting the service after the client bids $b_0, \ldots, b_i$. We have the client's probability of failure in the first bid is: $Pr\{\xi_0\} < (1 - 2^{-b_0})^{\frac{2^{b_0-1}L}{Z} - \tau s}$.

The adversary can continue to block b_1 only if the client cannot generate the bid b_1 in the following $(2^{b_1-1}L/Z - \tau s) - (2^{b_0-1}L/Z - \tau s) = (2^{b_1-1} - 2^{b_0-1})L/Z$ steps. This gives us: $Pr\{\xi_1\} < (1 - 2^{-b_0})^{\frac{2^{b_0-1}L}{Z} - \tau s} (1 - 2^{-b_1})^{\frac{(2^{b_1-1} - 2^{b_0-1})L}{Z}}$.

Following this line of reasoning, we can obtain the result of the theorem. □

References

1. The network simulator - ns-2. http://www.isi.edu/nsnam/ns/
2. Overview of cisco 7500 series router. http://www.cisco.com/en/US/products/hw/routers/ps359/prod_brochure09186a008009200c.html
3. Abadi, M., Burrow, M., Manasse, M., Wobber, T.: Moderately hard, memory-bound functions. In: Proceedings of the 10th Annual Network and Distributed System Security Symposium (2003)
4. Adkins, D., Lakshminarayanan, K., Perrig, A., Stoica, I.: Taming ip packet flooding attacks. In: Proceedings of Workshop on Hot Topics in Networks (HotNets-II). November (2003)
5. Adler, M.: Tradeoffs in probabilistic packet marking for IP traceback. In: Proceedings of 34th ACM Symposium on Theory of Computing (STOC-02) (2002)
6. Andersen, D.: Mayday: Distributed filtering for internet services. In: Proceeding of USITS (2003)
7. Anderson, T., Roscoe, T., Wetherall, D.: Preventing internet denial-of-service with capabilities. In: Proceedings of Workshop on Hot Topics in Networks (HotNets-II), November 2003
8. Argyraki, K., Cheriton, D.R.: Network capabilities: The good, the bad and the ugly. In: Proceedings of the 4th Workshop on Hot Topics in Networks, November 2005
9. Aura, T., Nikander, P., Leiwo, J.: Dos-resistant authentication with client puzzles. In: Proceedings of the Cambridge Security Protocols Worshop 2000. LNCS. Springer, Heidelberg (2000)
10. Internet Assigned Numbers Authority. ICMP type numbers. November, 2003. http://www.iana.org/assignments/icmp-parameters
11. Bellare, M., Rogaway, P.: Random oracle are practical: a paradigm for designing efficient protocols. In: Proceedings of First ACM Annual Conference on Computer and Communication Security (1993)
12. Bellovin, S.: Defending against sequence number attacks. In: RFC 1948, May 1996
13. Bellovin, S., Leech, M., Taylor, T.: The ICMP traceback messages. In: *Internet-Draft, draft-ietf-itrace-01.txt*, December 1999. ftp://ftp.ietf.org/internet-drafts/draft-ietf-itrace-01.txt
14. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
15. Boneh, D., Naor, M.: Timed commitments (extended abstract). In: Proceedings of Advances in Cryptology—CRYPTO'00. Lecture Notes in Computer Science, vol. 1880, pp. 236–254. Springer, Heidelberg (2000)
16. Burch, H., Cheswick, B.: Tracing anonymous packets to their approximate source. In: Proceedings of the 14th USENIX System Administration Conference, December 1999
17. Caida. Skitter. 2003. http://www.caida.org/tools/measurement/skitter
18. CERT. Computer emergency response team, cert advisory ca-2001-01: Denial-of-service developments. 2000. http://staff.washington.edu/dittrich/misc/ddos
19. CERT. Advisory CA-96.21: TCP SYN flooding and IP spoofing attacks. 24 September 1996
20. Crosby, S.A., Dan, S.: Wallach. Denial of service via algorithmic complexity attacks. USENIX Security (2003)
21. Dean, D., Franklin, M., Stubblefield, A.: An algebraic approach to IP traceback. In: Proceedings of Network and Distributed System Security Symposium (NDSS-01), February 2001
22. Dean, D., Stubblefield, A.: Using client puzzles to protect tls. In: Proceedings of 10th Annual USENIX Security Symposium (2001)
23. Theodore Diament, Homin K. Lee, Angelos D. Keromytis, and Moti Yung. The dual receiver cryptosystem and its applications. In: Proceedings of the 11th ACM conference on Computer and communications security. ACM Press, New York (2004)
24. Dietrich, S., Long, N., Dittrich, D.: Analyzing distributed denial of service attack tools: The shaft case. In: Proceedings of 14th Systems Administration Conference, LISA 2000 (2000)
25. Dittrich, D.: Distributed denial of service (ddos) attacks/tools resource page (2000) http://staff.washington.edu/dittrich/misc/ddos

26. Dwork, C., Naor, M.: Pricing via processing or combating junk mail. In: Brickell, E. (ed.) Proceedings of ADVANCES IN CRYPTOLOGY—CRYPTO 92. Lecture Notes in Computer Science, vol. 1328, pp. 139–147. Springer, Heidelberg (1992)
27. Feng, W.: The case for TCP/IP puzzles. In: Proceedings of ACM SIGCOMM Future Directions in Network Architecture (FDNA-03) (2003)
28. Feng, W., Kaiser, E., Luu, A.: Design and implementation of network puzzles. In: Proceedings of IEEE INFOCOM (2005)
29. Ferguson, P., Senie, D.: RFC 2267: Network ingress filtering: defeating denial of service attacks which employ IP source address spoofing, January 1998. ftp://ftp.internic.net/rfc/rfc2267.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc2267.txt
30. Franklin, M.K., Malkhi, D.: Auditable metering with lightweight security. In: Hirschfeld, R. (ed.) Proceedings of Financial Cryptography 97 (FC 97). Lecture Notes in Computer Science. Springer, Heidelberg (1997)
31. Garay, J.A., Jakobsson, M.: Timed release of standard digital signatures. In: Proceedings of Financial Cryptography (2002)
32. Geng, X., Whinston, A.: Defeating distributed denial of service attacks. IEEE IT Profes. **2**(4), 36–41 (2000)
33. Gligor, V.G.: Guaranteeing access in spite of service-flooding attack. In: Hirschfeld, R. (ed.) Proceedings of the Security Protocols Workshop. Lecture Notes in Computer Science. Springer, Heidelberg (2004)
34. Goldschlag, D.M., Stubblebine, S.G.: Publically verifiable lotteries: Applications of delaying functions (extend abstract). In: Proceedings of Financial Cryptography (1998)
35. Houle, K., Weaver, G., Long, N., Thomas, R.: Trends in denial of service attack technology. October 2001. http://www.cert.org/archive/pdf/DoS_trends.pdf
36. Ioannidis, J., Bellovin, S.: Implementing pushback: Router-based defense against ddos attacks. In: Proceedings of the Symposium on Network and Distributed System Security (NDSS-02) (2002)
37. Jin, C., Wang, H., Shin, K.G.: Hop-count filtering: an effective defense against spoofed traffic. In: Proceedings of ACM CCS (2003)
38. Juels, A., Brainard, J.: Client puzzle: a cryptographic defense against connection depletion attacks. In: Kent, S. (ed.) Proceedings of NDSS'99, pp. 151–165 (1999)
39. Keromytis, A., Misra, V., Rubenstein, D.: SOS: Secure overlay services. In: Proceedings of ACM SIGCOMM, August (2002)
40. Lemmon, J.: Resisting syn flood dos attacks with a syn cache. In: Leffler, S.J. (ed.) Proceedings of BSDCon 2002. USENIX, 2002, February 11–14
41. Li, J., Mirkovic, J., Wang, M.: Save: Source address validity enforcement protocol. In: Proceedings of IEEE INFOCOM (2002)
42. Mahajan, R., Bellovin, S., Floyd, S., Ioannidis, J., Paxson, V., Shenker, S.: Controlling high bandwidth aggregates in the network. CCR **32**(3), 62–73 (2002)
43. Meadows C.: A cost-based framework for analysis of denial of service networks. J. Comput. Secur. **9**, 143–164 (2001)
44. Merkle, R.C.: Secure communications over insecure channels. Commun. ACM **21**, 294–299 (1978)
45. Morein, W.G., Stavrou, A., Cook, D.L., Keromytis, A.D., Misra, V., Rubenstein, D.: Using graphic turing tests to counter automated ddos attacks against web servers. In: Proceedings of ACM CCS (2003)
46. Postel, J.: RFC 792: Internet Control Message Protocol, September 1981. ftp://ftp.internic.net/rfc/rfc792.txt
47. Qie, X., Pang, R., Peterson, L.: Defensive programming: Using an annotation toolkit to build dos-resistant software. In: Proceedings of the 5th OSDI Symposium, December 2002
48. Rivest, R.L., Shamir, A., Wagner, D.: Time-lock puzzles and timed-release crypto. Manuscript, 10 March 1996
49. Sanchez, L., Milliken, W.C., Snoeren, A., Tchakountio, F., Jones, C., Kent, S., Partridge, C., Strayer, W.T.: Hardware support for a hash-based IP traceback. In: Proceedings of the DARPA Information Survivability Conference and Exposition II, DISCEX'01 2001.
50. Savage, S., Wetherall, D., Karlin, A., Anderson, T.: Network support for IP traceback. In: Proceedings of ACM SIGCOMM, August 2000
51. Schnackenberg, D., Djahandari, K., Sterne, D.: Infrastructure for intrusion detection and response. In: Proceedings of the DARPA Information Survivability Conference and Exposition 2000, March 2000
52. Snoeren, A., Partridge, C., Sanchez, L., Jones, C., Tchakountio, F., Kent, S., Strayer, W.T.: Hash-based IP traceback. In: Proceedings of the ACM SIGCOMM, August 2001
53. Song, D., Perrig, A.: Advanced and authenticated marking schemes for IP traceback. In: Proceedings of IEEE INFOCOMM, April 2001
54. Stoica, I., Shenker, S., Zhang, H.: Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In: Proceedings of ACM SIGCOMM (1998)
55. Stone, R.: An IP overlay network for tracking dos floods. In: Proceedings of USENIX Security Symposium (2000)
56. Syverson, P.: Weakly secret bit commitment: Applications to lotteries and fair exchange. In: Proceedings of IEEE Computer Security Foundations Workshop (1998)
57. Touch, J.: RFC 1810: Report on MD5 performance, June 1995. ftp://ftp.internic.net/rfc/rfc1810.txt, ftp://ftp.math.utah.edu/pub/rfc/rfc1810.txt
58. von Ahn, L., Blum, M., Hopper, N., Langford, J.: Captcha: Using hard ai problems for security. In: Proceedings of Eurocrypt, pp. 294–311 (2003)
59. Wang, X.F., Reiter, M.: Defending against denial-of-service attacks with puzzle auctions. In: IEEE Symposium on Security and Privacy, May 2003
60. Wang, X.F., Reiter, M.: Mitigating bandwidth-exhaustion attacks using congestion puzzles. In: Proceedings of the 11th ACM conference on Computer and Communication Security, November 2004
61. Brent Waters, Ari Juels, J. Alex Halderman, and Edward W. Felten. New client puzzle outsourcing techniques for dos resistance. In: Proceedings of the 11th ACM conference on Computer and communications security. ACM Press (2004)
62. Yaar, A., Perrig, A., Song, D.: Pi: A path identification mechanism to defend against DDoS attacks. In: IEEE Symposium on Security and Privacy, May 2003. http://www.ece.cmu.edu/~adrian/projects/pi.ps
63. Yaar, A., Perrig, A., Song, D.: An endhost capability mechanism to mitigate DDoS flooding attacks. In: Proceedings of the IEEE Symposium on Security and Privacy, May 2004
64. Yau, D., Liu, C., Liang, F.: Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. In: Proceedings of IEEE International Workshop on Quality of Service (IWQoS-02) (2002)

Authors Biography

XiaoFeng Wang received his Ph.D. in Computer Engineering from Carnegie Mellon University in 2004. He joined Indiana University at Bloomington as an assistant professor in 2004.

His research interests span all areas of computer and communication security. Particularly, he is carrying out active research on system and network security (including malware detection and containment, countermeasures to denial of service attacks), privacy-preserving techniques and their application to critical information systems (such as medical information systems), and incentive engineering in information security.

Michael Reiter is the Lawrence M. Slifkin Distinguished Professor in the Department of Computer Science at the University of North Carolina at Chapel Hill (UNC). He received the B.S. degree in mathematical sciences from UNC in 1989, and the M.S. and Ph.D. degrees in computer science from Cornell University in 1991 and 1993, respectively. He joined AT&T Bell Labs in 1993 and became a founding member of AT&T Labs – Research when NCR and Lucent Technologies (including Bell Labs) were split away from AT&T in 1996. He then returned to Bell Labs in 1998 as Director of Secure Systems Research. In 2001, he joined Carnegie Mellon University as a Professor of Electrical & Computer Engineering and Computer Science, where he was also the founding Technical Director of CyLab. He joined the faculty at UNC in 2007.

Dr. Reiter's research interests include all areas of computer and communications security and distributed computing. He regularly publishes and serves on conference organizing committees in these fields, and has served as program chair for the flagship computer security conferences of the IEEE, the ACM, and the Internet Society. He currently serves as Editor-in-Chief of ACM Transactions on Information and System Security and on the Board of Visitors for the Software Engineering Institute. He previously served on the editorial boards of IEEE Transactions on Software Engineering, IEEE Transactions on Dependable and Secure Computing, and the International Journal of Information Security, and as Chair of the IEEE Technical Committee on Security and Privacy.