Mitigating DDoS Attacks with Micropayments

Ryan Moriarty
Courant Institute of Mathematical Sciences
New York University
rpm295@nyu.edu

***Abstract*—A primary concern in Internet security today is the threat of distributed denial-of-service (DDoS) attacks. This paper suggests the design, implementation, and analysis of a system aimed at mitigating these attacks. Like previous lines of work, the system aims to reduce the asymmetry between the resources needed to conduct an attack and those needed to absorb one. It does so by apportioning spare server capacity according to bids placed via a blockchain based smart contract.**

***Index Terms*—Denial-of-Service, Smart Contracts**

I. Introduction

Security was not one of the primary design goals of the early Internet. As the Internet has grown from a few hundred end hosts to several billion it has engendered a wide swath of unforeseen vulnerabilities.

A. DDOS Attacks

One particularly detrimental class of attacks resultant from this rapid growth are distributed denial-of-service (DDoS) attacks, in which an adversary uses a large number of compromised machines to inundate a network resource in order to prevent its legitimate use. Lax security measures in Internet of things devices have enabled attackers to amass large robot networks which they can then command to overwhelm unknowing targets. Data sent from these devices appears indistinguishable from legitimate traffic, preventing most means of filtering.

Although DDoS attacks can happen at any network layer, this paper focuses on those at the application-level. These attacks often exploit asymmetries in the amount of work performed by different parties executing certain protocols. Examples include HTTP GET requests or SSL handshakes, in which short client requests invoke a server to expend a disproportionate amount of resources.

Much work has been dedicated towards DDoS prevention and mitigation techniques. Some common techniques include ingress filtering (which can be difficult), over-provisioning (which is wasteful), and black hole routing (which punishes legitimate users as well as malicious ones). Another line of research aims to balance the asymmetry between client and server workload by requiring clients expend a certain resource in order to guarantee service. Resources that have been suggested have included computation [4], memory [1], bandwidth [12], and money [10]. The hope being that allocating service in proportion to one of these metrics, as opposed to requests per second, will result in a greater allotment for legitimate clients.

B. Proposed Solution

This paper contributes the design, analysis, and implementation of a blockchain-based DDoS mitigation system in which server resources are auctioned off under times of duress. When the server becomes oversubscribed, it will begin asking clients to bid for service, apportioning spare capacity to winning bidders. Both client and server interact with a smart contract, which is a computerized transaction protocol that executes the terms of a contract. The contract runs on the Ethereum blockchain, which is a decentralized ledger capable of executing arbitrarily complex code. The Ethereum network has a native currency called ether which is used to pay for the computational resources executed over the network. Clients will use the smart contract to place their bids, while the server will use the smart contract to identify new bidders and extract fees from winning bidders.

The auction will be conducted using a thinner, an idea adopted from [12], which is a front-end that filters incoming requests down to a level the server can handle. It does so by placing verified client requests into an auction pool which is continuously drawn from at a rate proportional to the server's capacity. Winning requests are sent through to the server which uses HTTP cookies to ensure a user has a sufficient balance to cover their requests. At the end of a completed session, the server calls the smart contract to deduct the appropriate fees from the user's balance.

II. Design

A. Rationale

Using money as a scarce resource should act as the greatest deterrent to an adversary because, unlike other proposed resources, it doesn't necessarily scale with the number of devices under his control. Each device an adversary is able to compromise adds to his aggregate compute power, memory, and bandwidth, but his budget remains more or less constant. Compromised devices could potentially confer monetary gain via extortion or cryptocurrency mining, however, the relative resource gain per device is not nearly as linear a relationship as the others.

Furthermore, the size of the adversary's botnet is nullified since his optimal strategy for continuing to block traffic is to bid as much as possible on just enough devices to saturate the server's excess capacity, e.g. if the server's capacity is 1000 requests per second, he would be better off bidding \$1 across 1,000 devices than \$.01 across 100,000 devices. Thus, his goal

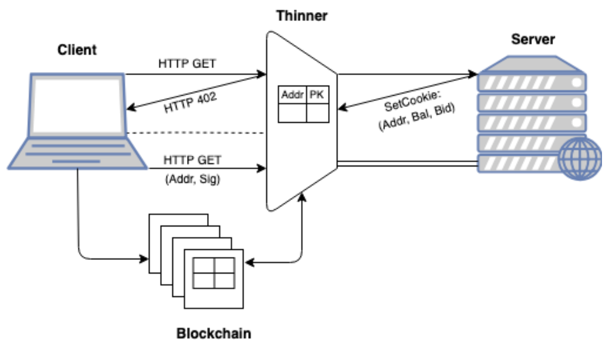

Figure 1: The architecture of the proposed system

becomes to allocate a resource as efficiently as possible as opposed to producing the highest aggregate amount of it.

The counterargument to using money is that people don't like to pay for things, especially on the internet where there is a certain expectation that content should be free. However, the monetary cost on the legitimate user will be minuscule, likely fractions of a penny per request. The real cost will be the time it takes to initially fund the smart contract and the wait time before the transaction is added to the blockchain (roughly 8 seconds on average). However, this is only a one time occurrence and a single smart contract can be made to work across multiple websites.

An additional downside is that the client will need to have a blockchain enabled browser like Opera or a plug-in like Metamask in order to interact with the smart contract. Thus, it is likely the system will only be viable for websites that already require this functionality. In some sense the adoption of the system assumes a future in which blockchain based web micropayments are ubiquitous and have replaced the ad-based model as the dominant source of revenue for content providers. Under this scenario, clients will already have a funded cryptocurrency account linked to their browser and will be accustomed to paying for content.

B. Architecture

The design of the system is shown in Figure 1. To understand how the mechanism would work in practice, consider the following attack scenario. A user, Alice, is trying to access a website under a HTTP GET attack by a malicious adversary.

Since the web server is receiving more traffic than it can handle, the front-end will kick in to begin filtering the number of requests that get through. Upon Alice's first attempt to access the site, the thinner, following the logic shown in Listing 1, will see that she has not provided a valid address and will send her a request for payment with a link to the ServerAuction smart contract shown in Listing 2. Alice may then call the contract's $newBidder()$ function specifying a bid and a public key along with a deposit in order to fund the account.

After Alice's transaction is placed into a block and appended to the blockchain, it will emit an event, which the thinner listens for. The thinner then extracts Alice's Ethereum address, balance, bid, and public key from the event and enters it into a table along with a bit which signals whether or not Alice is currently waiting for service. An example entry is shown in Table 1.

Alice then sends a new HTTP request to which she appends

```
upon incoming request:
if (address in table):
    if (verify(sig, pk) == true):
        waitBit = 1
    else:
        deny request
else:
    send HTTP 402 response

once every 1/C seconds:
send request corresponding to max(waitBit || bid) to the server
```

Listing 1: Thinner Functionality

```
contract ServerAuction {

    event Deposit(
      address from,
      uint depos,
      uint bid,
      uint256 pubKey
    );

    struct Bidder {
      uint balance;
      uint bid;
      uint256 pubKey;
    }

    address internal sellerAddress;
    mapping(address => Bidder) private Bidders;

    constructor(address sellerAddress) public {}

    function newBidder(uint depos, uint bid, uint256 pubKey) public {
      emit Deposit();
    }

    function addToBalance() public payable {}

    function changeBid(uint bid) public {}

    function changePubKey(uint pubKey) public {}

    function deductFromBalance(address cust, uint price) public {}

    function withdraw(uint amount) public {}
}
```

Listing 2: Auction Smart Contract API

```
cost = bid + (currentTime − startTime) ∗ timeFee
if (balance − cost >= 0):
    balance −= cost
    startTime = currentTime
    send requested resource
else:
    send HTTP 402 response
```

Listing 3: Server Logic Upon New Requests

address	balance	bid	pubKey	waitBit
0xBBA0f…	1	.0001	0xF62EF4…	0

Table 1: Thinner Entry

her Ethereum address and a signature. When this request is received by the thinner it will confirm that the address is in its table, verify that the signature is valid, and set the waitBit of Alice's entry to 1.

Once every $\frac{1}{C}$ seconds, where C equals the server's capacity, the thinner will send the request with the highest bid through to the server. As long as the adversary isn't continuously absorbing spare server capacity with bids higher than Alice's, her request will eventually get through to the server.

The server then sets a cookie with Alice's balance and bid which is used to ensure that Alice has sufficient funds for each resource she requests, as shown in Listing 3. At the end of the session, the server passes the total cost of the session back to the thinner, which then calls the contract's $deductFromBalance()$ function to extract the required payment from the client.

C. Additional Considerations

1) CAPTCHA: At first glance it may seem that the steps needed to fund an Ethereum account and place a bid through the smart contract confers the additional benefit of acting as reverse Turing test. However, an adversary could create these accounts centrally and then subsequently delegate them to different machines. Each of these bots simply would need to relay this information along with a digital signature in its web request to the server.

2) Authentication: Public and private keys are needed in order to prevent an adversary from spoofing valid addresses. The system can harness the asymmetric key infrastructure already in place on the Ethereum network.

3) Thinner: We assume the thinner has enough processing capacity to absorb a full DDoS attack. This is not unrealistic since the amount of work performed by the thinner per-request is minimal compared to that performed by the server. Periodically, the thinner may have to remove entries from its table in order to prevent an adversary from exhausting its storage capacity. It can do so easily by removing stale entries or those with the lowest bids.

4) Bidding Denominations: Bidding denominations should correspond to the amount of work performed by the server. This could be dollar per byte retrieved, dollar per sever seconds consumed, or dollar per request scaled by some multiplier dependent on the difficulty of the request.

5) Slowloris: The server also keeps track of the duration of each session. A 'slowloris' attack attempts to hold web server connections open as long as possible by sending an incomplete request and subsequently sending partial information in order to avoid being timed out. To deter this, the server deducts a fixed rate per second from the user's balance.

III. ANALYSIS

In this section we will try to estimate the cost imposed on an adversary attempting to attack our proposed system. We will use the fraction of legitimate requests served as a performance metric, as this seems like the most relevant concern from an end-user's perspective.

A. Theoretical Cost Analysis

We define the server's capacity as C requests per second, the aggregate capacity from good clients as G and the aggregate capacity from bad clients as B. We assume $G < C < B$, a scenario in which the first inequality doesn't hold corresponds to a flash crowd and that in which the second doesn't hold corresponds to a manageable level of traffic. Without the thinner, the percent of good client requests being fulfilled is equal to $\frac{C}{G+B}$, i.e. the servers capacity divided by total client capacity. To assess the quality of service with the auction mechanism in place, we need to know A, the amount the dollars the adversary is willing to spend per second. We will further assume that the bids by good clients resemble a normal distribution and that the adversary bids according to the following theorem.

Theorem. *With a fixed budget, the adversary maximizes the amount of capacity captured by bidding evenly over all devices.*

Proof. The adversary blocks all requests to the server that are below his lowest winning bid. He can maximize the amount of requests blocked by maximizing his minimum bid or by bidding evenly across all devices. □

We can then estimate the adversary's amount bid per request, D. Since the attacker will absorb any remaining bandwidth at this price point, no request with a bid less than D will be serviced. If we denote the area of the distribution below this price point as f, the percent of good client requests fulfilled will equal $1 - f$. The amount of capacity captured by the adversary will equal $C - G + f \cdot G$, i.e. the capacity unused by G plus the amount captured from G. D can then be written as $\frac{A}{C-G+f\cdot G}$ and the percent of G requests served can be solved for with the following equation:

$$1 - f = 1 - P\left(z < \frac{\frac{A}{C-G+f\cdot G} - \mu}{\sigma}\right) \qquad (1)$$

Figure 2 depicts the fraction of service to good clients prior to activating the thinner. To derive a similar plot after the auction we would have to arbitrarily fix the adversary's budget. However, we can solve for the budget required for the adversary to maintain his level of DoS by fixing the value of C, G, B, μ, and σ, and setting $\frac{C}{G+B}$ equal to the right hand side of (1). The results are displayed in Figure 3 as a function of G and B.

The above analysis assumes good clients only place a single bid without rebidding. In reality, bidding may be a dynamic process in which users could deploy a variety of strategies including probing for an acceptable price by systematically

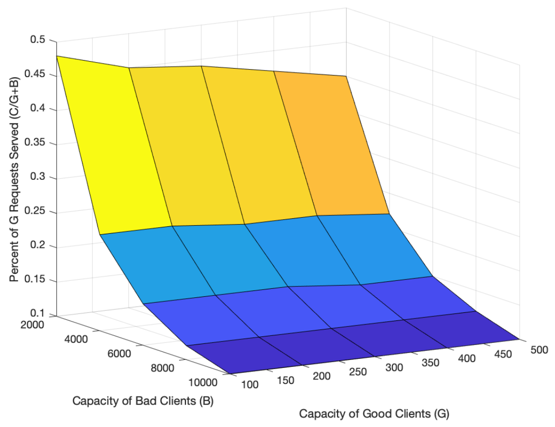

Figure 2: The fraction of good client requests served prior to the auction as a function of G and B, for a fixed C of 1000

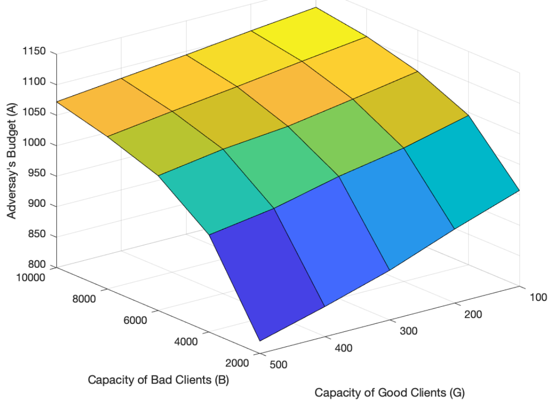

Figure 3: Budget, in multiples of the mean bid, needed to maintain the level of denial of service prior to the auction as a function of G and B, for a fixed C of 1000

increasing their bids. Since legitimate clients would have the added freedom of altering their bids while the adversary's optimal strategy remains fixed, we can be confident that f acts as an upper bound on the fraction of legitimate requests blocked by the attacker.

B. *Heuristical Cost Analysis*

As shown in Figure 3 the cost of an attack per second is going to be roughly $C \cdot \mu$ regardless of the capacity of good and bad clients. If we assume the average web server can handle 1,000 requests per second and that the average user would pay \$.01 per webpage, e.g. the user pays \$50 per month for internet service and visits 5,000 webpages in that time period, it would cost an attacker \$10 per second to perform an average DDoS attack. Using 15 minutes as a conservative estimate of the average attack length, the total cost of the attack would be roughly \$10,000. This is likely far too costly for all but the most well-resourced adversary.

Fraction of G requests served	.93	.7	.33	.08
\$ if bids are normally dist.	979.0	1207.4	1498.0	1775.8
\$ if bids are identical	976.1	1204.0	1495.1	1772.6

Table 2: Adversary's cost as a function of bidding strategy

% of rebidding clients	0	.01	.1	1	10
Fraction of G requests served	.5	.51	.57	.87	.99

Table 3: Rebidding clients

IV. EXPERIMENTATION

A. *Baseline*

We performed a series of experiments using an auction simulator written in Python. The program, given parameters C, G, B, D, μ, σ, and the number of trials, initializes a set of bids drawn from a normal distribution. Each iteration of the auction compares the maximum value in this set to the value of D, if the value is larger, it is removed from the set and a good client is considered to have won. Good clients appear randomly at each iteration with probability $\frac{G}{C}$.

Our experiments revealed that the distribution of good client bids converges to the area below the amount bid by the adversary. This makes sense since requests are being removed from the distribution above D at a rate of C but are only being added to at a rate of $(1-f) \cdot G$, which by assumption is strictly less than C.

After a significant number of iterations, the fraction of good requests fulfilled is calculated and compared to the analytically calculated result. Our results confirmed the theoretical analysis presented in the preceding section.

B. *Adversary's Bidding Strategy*

We also simulated the effect of bad clients bidding according to a normal distribution. Upon each iteration, if the maximum bid in the auction pool belongs to a bad client it is removed and replaced with a new bid from the same distribution. Afterwards, we calculated the cost needed to perform the attack as well as the percent of legitimate requests blocked and compared it to the cost required to block that fraction of traffic if a constant bid was used. Our results showed that an adversary saves around .2% on average by bidding evenly across his devices then by bidding according to a normal distribution, as shown in Table 2. Because the cost reduction is slight, an argument can be made that the adversary would be better off varying his bids in order to disguise which requests are his.

C. *Rebidding Clients*

A simulation of a dynamic bidding environment was also conducted in which after each second, some fraction of the good clients increased their bid by one standard deviation. This corresponds to the probing type strategy mention in section III A. The results are displayed in Table 3, which show that if legitimate clients adopt a dynamic bidding strategy they can drastically reduce the adversary's effectiveness.

Fraction of G who bid	.2	.4	.6	.8	1
Fraction of G requests served	.14	.28	.42	.56	.7

Table 4: Non-bidding clients

Seconds until termination	.5	1	2	5
Fraction of requests effected	.092	.044	.026	.005

Table 5: Requests effected as a function of seconds until request termination

D. Non-bidding Clients

We also examined the impact that would result from a percentage of the good clients choosing not to bid at all. This could be due to the cost or inconvenience of bidding. As shown in Table 4, we found that the percent of good requests served declined linearly with the percentage of non bidding clients. This implies that the effectiveness of the system depends on the fraction of legitimate clients who choose to bid for service.

E. Request Termination

Finally, we considered the effect of clients terminating their requests after a certain wait time. Our results showed that this did not significantly effect the adversary's effectiveness. Experiments suggested that clients tend to either get service immediately ($\approx .2$ seconds) or not at all, depending on their bid amount. Table 5 displays the percent of clients who would have gotten service but instead canceled their request. We found that the majority of clients who terminated their requests due to long wait times would not have gotten service anyway.

V. Implementation

We implemented a prototype of the system using an Ethereum based smart contract written in Solidity and deployed on the Ropsten testnet, as well as a thinner written in Python and deployed on an EC2 server. The smart contract implements the $newBidder()$ and $deductFromBalance()$ functions shown in Listing 2.

Some components of the thinner were simplified for ease of implementation. UDP was used instead of HTTP to simplify the payload inspection process. A password based scheme was used for verification purposes as opposed to an asymmetric system and connections were assumed to be non-persistent.

The thinner concurrently listens for UDP packets arriving on a specific port as well as events emitted from the smart contract. When a deposit event is heard, the relevant fields are extracted from the event's data field and entered into a dictionary similar to the one shown in Table 1.

When a new packet arrives, the payload is inspected. The server assumes the first 20 bytes correspond to the user's Ethereum address and the remaining bytes represent the user's password. If the payload is malformed or the address is not currently in the thinner's table, the server replies with a request for payment and the address of the smart contract. Otherwise, the password is hashed and compared to the challenge value stored in the table. If the values do not match, the server sends a reply informing the client their password is incorrect. Finally, the client's balance is checked to see if it greater than their bid amount, if not, the server responds with an insufficient balance message. Otherwise, the thinner adds the request to a priority queue sorted in descending order by bid amount and the user is informed as such.

A thread continuously pops from the request queue, sleeping for $\frac{1}{C}$ seconds between iterations. The destination corresponding to the popped request is then sent the desired resource, in our case a secret string. The $deductFromBalance()$ method is called with their bid amount, which is also subtracted from the user's balance listed in the thinner.

The thinner was evaluated with a client program also written in Python which tested each of the thinner's conditional branches, ensuring the intended functionality was provided.

VI. Discussion

A. An Analogy

The system is meant as a free-market solution which allocates service to those who desire it most. An analogy could be made to auctioning off tickets to a sporting event, a determined adversary could purchase a large number of tickets preventing fans from attending. However, his cost per ticket would need to be at least the amount that any of these fans are willing to pay. If the adversary's utility from denying service to a fan is greater than the utility the fan derives from that service, then this is an acceptable outcome from a free market perspective.

B. Flash Crowds

Since the system is client agnostic, there will be no clear cut way to distinguish between a DDOS attack and flash crowd. The thinner will activate in either scenario and those that value the service most will be given access to it, regardless of whether that value is derived from the service in itself or as a means of denying the service to someone else.

C. Reimbursement

It's worth mentioning that the mechanism is not meant to generate revenue. Any revenue gained under a DDOS attack or flash crowd will likely be less than the reputational cost resultant from legitimate users being unable to access the site. Although it may seem fair to partially reimburse customers after an attack, it is difficult to assess which payments came from legitimate users. Simply refunding everyone after an oversubscription would defeat the purpose of the auction and allow an adversary to conduct attacks without any downside risk. Reimbursing only clients who have been verified may be beneficial since it is unlikely an adversary would be able to verify enough accounts to make the cost of executing an attack worthwhile.

VII. Related Work

The first resource based defense scheme was presented by Dwork and Naor in [4], which required clients to extract square roots modulo primes as a means of combatting spam. Back [3] built on this work, proposing a CPU proof-of-work cost function as a denial-of-service prevention mechanism. Tangentially, this scheme was adopted by the creator of Bitcoin, the first blockchain based cryptocurrency and precursor to Ethereum, as a means of achieving consensus on the current state of the blockchain. Abadi [1] proposed using memory as opposed to CPU cycles to narrow disparities across computer-systems. Stravou [10] proposed a micropayment DDOS prevention scheme using a centralized clearing center. Lastly, Walfish proposed using bandwidth in [12], which inspired several of the ideas presented in this paper. Their scheme encouraged clients to bid for service by sending an increased amount of bandwidth, with the amount of bandwidth sent acting as bid in a virtual auction.

VIII. Conclusion

DDOS attacks are a serious concern for network administrators, evidenced by the 600k device Mirai botnet in late 2016 and the 1.3 Tbs attack on Github in 2018. This paper proposed a micropayment-based DDoS mitigation system, in which server resources are allotted to the highest bidder under times of oversubscription. Theoretical and heuristical analysis were presented which showed the financial cost imposed on an attacker would be sufficiently burdensome to deter all but the most well financed adversaries. The results were confirmed by experiments under both static and dynamic bidding environments. An interesting line of further work would be to see how the bidding process may unfold under an attack scenario with real humans and real financial interests.

A simplified prototype of the system was built proving the viability of a real-world deployment. The primary drawback of the system at the moment is the unfamiliarity of the general public with cryptocurrencies and the lack of blockchain integration into most web browsers. We hope that this paper serves as an example of the utility of blockchain based micropayments and encourage further research into ways in which digital currencies can be used to improve the Internet experience.

Acknowledgment

Thank you to Anirudh Sivaraman for advising this project from its inception.

References

[1] Abadi, M., Burrows, M., Manasse, M., and Wobber, T.. 2005. Moderately hard, memory-bound functions. ACM Trans. Internet Technol. 5, 2 (May 2005), 299-327. DOI=http://dx.doi.org/10.1145/1064340.1064341

[2] Antonakakis, M. et al. Understanding the mirai botnet. In Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17), Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, Berkeley, CA, USA, 1093-1110.

[3] Back, Adam. (2002). Hashcash - A Denial of Service Counter-Measure.

[4] Dwork C., Naor M. (1993) Pricing via Processing or Combatting Junk Mail. In: Brickell E.F. (eds) Advances in Cryptology — CRYPTO' 92. CRYPTO 1992. Lecture Notes in Computer Science, vol 740. Springer, Berlin, Heidelberg

[5] Juels, A. and Brainard, J. 1999. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In Proceedings of the Conference on Network and Distributed System Security Symposium (NDSS).

[6] Leiwo, J., & Zheng, Y. (1997). A Method to Implement a Denial of Service Protection Base. ACISP.

[7] Mirkovic, J. and Reiher, P. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. SIGCOMM Comput. Commun. Rev. 34, 2 (April 2004), 39-53. DOI=http://dx.doi.org/10.1145/997150.997156

[8] Rivest R.L., Shamir A. (1997) PayWord and MicroMint: Two simple micropayment schemes. In: Lomas M. (eds) Security Protocols. Security Protocols 1996. Lecture Notes in Computer Science, vol 1189. Springer, Berlin, Heidelberg

[9] Sekar, Vyas & Duffield, Nick & Spatscheck, Oliver & E. van der Merwe, Jacobus & Zhang, Hui. (2006). LADS: large-scale automated DDoS detection system. 171-184.

[10] Stavrou, A., Ioannidis, J., Keromytis, A.D., Misra, V., & Rubenstein, D. (2004). A Pay-per-Use DoS Protection Mechanism for the Web. ACNS.

[11] Szabo, N..“Smart contracts, 1994“.

[12] Walfish, M., Vutukuru, M., Balakrishnan, H., Karger, D., and Karger, D., Shenker, S. 2006. DDoS defense by offense. SIGCOMM Comput. Commun. Rev. 36, 4 (August 2006), 303-314. DOI: https://doi.org/10.1145/1151659.1159948

[13] Wang, X. & Reiter, M.K. Int. J. Inf. Secur. (2008) 7: 243. https://doi.org/10.1007/s10207-007-0042-x