An Analysis of Flow Identification in QoS Systems

Michael Piatek
Department of Mathematics and Computer Science
Duquesne University
piatek@mathcs.duq.edu

Abstract

This paper presents the confounding socket, a mechanism that makes real-time flow classification computationally infeasible by masking the protocol structure of any application's network traffic. Current commercial QoS systems rely on well-known application characteristics such as port ranges and protocol characteristics to provide admission control. The confounding socket invalidates these criteria by encrypting all network traffic beginning with an implicit and unrecognizable key exchange phase. Experiments suggest that translated traffic is effectively random. These results demonstrate a fundamental weakness in current QoS flow classification methods.

1 Introduction

Demand for internet quality of service (QoS) has been growing rapidly in recent years, especially at the network boundary of large organizations. Such organizations often have an internet link responsible for servicing hundreds or even thousands of internal network nodes. Network administrators are tasked with meeting the bandwidth and latency requirements of critical network services. Recent, explosive growth of bandwidth hungry peer-to-peer applications has made this especially difficult [5, 18].

One method of providing QoS is resource reservation based on admission control, which excludes or controls the behavior of certain flows when the network becomes overloaded. To provide meaningful admission control, the network must first know something about the traffic characteristics of a particular flow. The internet currently provides no admission control and only one class of "best-effort" service for all traffic. Because of this, most commercial QoS devices are found at the boundary between the internet and the private local networks of large organizations. Admission control for these devices is based on classification of network flows by protocol and the discretion of system administrators, who define traffic classes for identified protocols.

Admission control on the network boundary has advanced with the continued efforts of peer-to-peer software groups to avoid protocol identification and associated bandwidth shaping. This is most clear in the transition from the centralized index search model of Napster [15] to the decentralized, broadcast flood search of the Gnutella network [8]. Classifying Napster traffic is trivial because of its use of a well-known port and central database host. The Gnutella network, however, is designed specifically to eliminate both port and host dependencies. This model has been reproduced in several other modern peer-to-peer networks [10, 14, 4]. The popularity of these new networks has driven administrators to adopt new QoS systems that identify traffic at the protocol, or layer 7, level of the OSI network model. These devices first examine raw network traffic to identify a flow, and then use low-level TCP flow control mechanisms to adjust the rate of data transfer. Such systems include Packeteer's PacketShaper [17], Cisco's Network-Based Application Recognition [3], and Sitara Networks QoSWorks [16].

Our goal is to explore the long-term viability of this type of traffic analysis as a means to identify network flows in spite of direct circumvention efforts by bandwidth intensive services. The effectiveness of current boundary QoS devices relies on each vendor's ability to develop deterministic identifiers for any network flow that might require guaranteed or restricted resource allocation. We present the confounding socket, a translation layer capable of masking the protocol structure of any TCP flow. This technique represents a weakness in current layer 7 traffic analysis that will eventually cripple its ability to function as a basis for providing QoS.

The remainder of this paper is organized as follows. We examine related literature in Section 2. In Section 3, we detail the translation process of the confounding socket and analyze its resistance to identification. In Section 4, we apply our technique to the Limewire [11] Gnutella servent[1] and examine its effects. Section 5 explores the social and technological implications of our technique. Section 6 suggests possibilities for future research and applications. Finally, Section 7 summarizes the significance of this paper.

2 Related work

The problems of deterministic flow identification have received little attention in literature. This is likely due to its very recent explosion in use to combat peer-to-peer bandwidth consumption. Relevant analysis has focused on methods of identifying flow needs by observing traffic properties rather than protocol structure. For examples of this technique, see [7, 2].

The need for admission control and the implications of implicitly supplied service were treated by Shenker in [20]. Particularly, he speculated that the inclusion of application specific knowledge at the router level would inhibit development of new applications. We further build on this idea in Section 5. The use of the confounding socket by unbounded bandwidth consumers such as peer-to-peer applications will force administrators to give all unidentified traffic identical, low priority service. This forces new applications to handle poor service until flow identifiers are developed for their particular protocol.

3 The Confounding Socket

Our goal of concealing structure immediately suggests the use of key-based encryption of some sort such as TLS [6]. However, classic methods of providing encrypted network transport have key exchange phases that are well defined and provide a means for identification during the connection buildup phase. What is called for, then, is a method of handshaking an encrypted connection and performing key exchange that cannot be recognized by an intermediary with unrestricted access to network traffic, but is recognizable by each endpoint.

Our approach is to use an implicitly exchanged session key. Each peer shares an identical pool of cipher keys of size N. These keys are public and distributed with the application. Prior to creating an outgoing connection, a session key is chosen at random from the shared pool to be used throughout the entire connection. A specific hello message and random integer are encrypted with this key and immediately sent to the remote host upon successful TCP connection. The remote host then attempts to decrypt the message using each key in the shared pool. When a decryption attempt is made that reveals the specific hello message at the expected location, both peers know the session key. This process is further detailed in Figure 1.

Once handshaking is complete, all subsequent traffic is encrypted using the session key. Every packet is preceded by a header packet, also encrypted with the session key, which contains the following packet's size and a random integer to salt the header. Although this technique is applicable to any keyed encryption method, our implementation uses RC4 so that the end of an encrypted packet can be determined during buffered reading using provided sizing information, as encrypted RC4 traffic is identical in length to its associated plain-text. We include another random integer with each message's content portion to salt the encrypted data and force decryption to recover protocol structure. Otherwise, pattern-matching identification would be effective by matching against the N well known structures arising from each of the session keys. Also, each output buffer flush must be divided into at least two randomly sized portions so as to avoid recognition by packet size if a protocol repeatedly uses fixed size control messages. For a more information on confounding translation, see Appendix A.

Observe that the session key has never been explicitly transferred during connection buildup. To recover the session key, the bandwidth shaping device would have to perform the same linear key search as each endpoint. The main observation here is that the collective computing power of all the intranet peers outweighs the computing power of the gateway device. Further, the identification problem at the network edge is compounded by the uncertainty of which flows are attempting to use the confounding translation technique. Flow identifiers exist for only a small subset of all network applications. However, in order to check for a flow employing confounding translation, all flows that do not have predefined identifiers would have to be checked with each potential session key. Peers have the advantage of knowing which

[1]Peer-to-peer applications are often referred to as *servents* because they function as both *serv*er and cli*ent*.

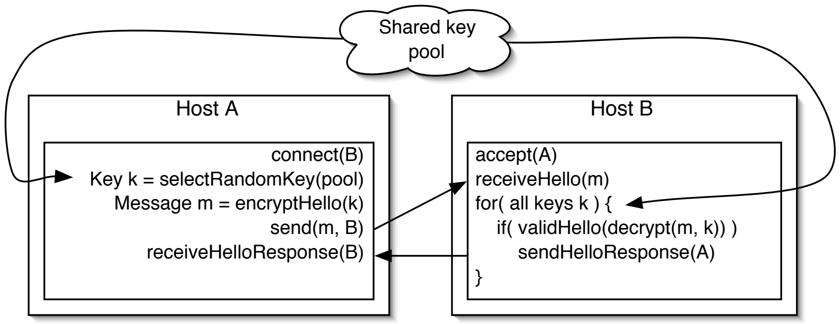

Figure 1: The implicit key exchange during connection buildup

connections they intend or expect to use confounding translation.

Although confounding translation does not provide long-term security by any means, as the shared key pool is public, recall that a confounding flow need only circumvent *real-time* flow identification. Because a bandwidth shaping device is generally responsible for serving many hosts, so long as the shared key pool is large enough and network activity is high enough, checking for confounding streams will be computationally infeasible. Also, in order to scale with increasing computation power of shaping devices, application developers need only increase the size of the shared key pool.

4 A case study: Gnutella

4.1 Motivation

Because the performance and size overhead of the confounding socket depends greatly on the type of traffic it is transmitting, theoretical analysis is ineffective. We evaluate the performance of our technique empirically within the context of the Gnutella overlay network. Gnutella was arguably the first decentralized file-sharing network to gain wide popularity, particularly after the collapse of the Napster network. The scalability problems of the initial Gnutella protocol are well documented [19], generally making it one of the most restricted file-sharing protocols.

Unquestionably, peer-to-peer applications have the most incentive to adopt the confounding socket, as they are certainly the most affected by bandwidth shaping. Also, their popularity is dependent on a large user base of high-speed peers, often found at universities, to provide the majority of network accessible files [1]. However, it is precisely these peers that are generally subject to the most stringent bandwidth controls.

Peer-to-peer networks are also relatively easy to migrate to alternate transport methods because of their structure. In order to search a Gnutella network, or most peer-to-peer networks in general, a client need only be connected to one host. Consider a Gnutella network consisting of both confounding and normal peers where confounding peers can communicate with both types. We term such peers gateway nodes, as they provide a connection path for confounding peers unable to connect to normal hosts. Peers restricted by layer 7 throttling or blocking can use the searching and routing infrastructure of the entire network via gateway nodes to locate other confounding nodes. This observation suggests a practical migration scheme for any overlay network to move to the confounding socket.

4.2 Simulation methodology

We conducted our analysis on a modified Limewire [11] servent because it is the most actively developed open-source Gnutella client that implements the enhancements to the protocol made by the development community since the introduction of the original servent. The confounding socket sits between Limewire's connection and server classes and the underlying Java socket interface. This implementation made integration with Limewire simple, requiring the alteration of only socket and server socket creation code. The remaining confounding translation processes is entirely transparent to the application code.

To precisely compare the confounding socket to normal transport, exact Gnutella traffic was necessary. We used Holger Luemkemann's Gnutellasim

Message type	Frequency
Query request	60%
Query response	25%
Ping reply	10%
Ping	5%
Push request	<1%

Table 1: Message frequencies

Top hosts	Percent of total
1%	37%
5%	70%
10%	87%
15%	97%
20%	98%

Table 2: Sharing rate

utility [12] to crawl the real network and collect hosts, shared files, and queries. This utility uses the Limewire core for network connectivity, and its behavior as a servent is identical. These output files were then used to model a virtual Gnutella network using both the confounding transport output streams and normal, pass-through output. Protocol messages were generated according to the average frequencies gathered by the crawler (see Table 1). Push requests occurred so infrequently that they had no impact on our analysis, so we ignored them during traffic generation. Also, files were distributed among the simulated hosts according to the sharing statistics gathered by Adar and Huberman [1] (shown in Table 2).

4.3 Results

Our traffic generation trial produced 46,944 messages. The difference in raw traffic data size between confounded and normal transit was 1,098,924 bytes. Limewire output has two phases after connection buildup. First, the Gnutella header is written to the output stream and then the associated payload, with the exception of the ping message that contains no payload. Confounding sockets incur an overhead of 12 bytes per output write. The difference is accounted for by presuming all messages to incur 24 bytes of overhead for the two phased write and subtracting the nonexistent payload overhead of 2,311 ping messages. $46944 * 24 - 2311 * 12 = 1098924$. The confounding connection buildup also adds 169 bytes of additional overhead.

Also, we tested CPU utilization by transferring large files among hosts in a local Gnutella network. In a five node fully connected network, each 733MHz Pentium III client exhibited bandwidth bounded network transfer on a full duplex 10-megabit ethernet network. RC4 key strength was 192 bits. Processor utilization never exceeded 50%, suggesting that confounding transfer is feasible on recent commodity hardware. Further, simulation on a 10-megabit link is conservative, as the confounding socket is most likely to be used across WAN links that rarely approach LAN bandwidth availability.

The confounding socket intends to mask protocol structure. A reasonable metric for judging its success is randomness. In other words, if a stream has no structure, it should pass tests for randomness. We employed Marsaglia's well-known Diehard battery of tests for randomness [13] to analyze output from our traffic generator. As expected, normal Gnutella traffic unequivocally fails all of the performed tests. Confounded traffic, however, performs well enough to be considered a reasonable random generator. Our confounded sample contained 13,155,875 bytes of traffic and passed the series. (See http://www.cs.duq.edu/~piatek/qos/random/ for the results of the Diehard tests.)

5 Implications

Given the tendency of peer-to-peer developers to rapidly adopt any change that will improve user participation or circumvent bandwidth controls, widespread use of the confounding socket seems certain, at least among these types of applications. More fundamentally, this adoption will emphasize the basic vulnerability in flow identification represented by our technique. A posteriori flow classification is fundamentally flawed. The prolific exploitation of this weakness will create a massive surge in unidentified bandwidth consumption. We consider four possible solutions to this problem.

First, network administrators may simply change their QoS policies to place all unidentifiable flows in a low priority traffic class. This will certainly solve the immediate bandwidth usage problem, but it will do so at the cost of user satisfaction. Current policies generally rely on large bandwidth consumers being readily identifiable, allowing for targeted control.

Identification mechanisms need only be developed for such applications, leaving all other network traffic to fall into the unidentified and often best-effort class of service that generally takes up little bandwidth. Notice that placing large bandwidth consumers in the same traffic class forces small consumers to compete equally with large, resulting in poor or unusable performance for former. The solution to this subsequent problem that is possible within the capabilities of existing architecture is to develop bandwidth identifiers for *all* network protocols, a daunting and boundless task. This would provide implicit identification of protocols hiding beneath the confounding socket by excluding all others.

Another solution involves changing network architecture. Consider an example of a university's residential bandwidth management. Rather than provide service guarantees to specific services, guarantees can be made on the basis of connection location by subnetting residential, faculty, and research hosts and giving specific subnets a guaranteed share of resources. Certainly this would provide different tiers of service to specific users. However, within each of the widely allocated traffic classes, behavior is still best effort. More acute service policies may be desired or required.

Third, outside the realm of technical methods, pricing mechanisms are becoming common. Tiered pricing, in which users are monitored for total bandwidth use regardless of type, is gaining popularity in the commercial ISP community to deter wasteful use [9]. However, this solution only makes sense when network access can be tied to specific users. On shared corporate machines or in a common use lab, attributing traffic use to specific users is much more difficult.

It is worth mentioning that the three aforementioned solutions all stray from the fundamental goals of QoS because they do not solve the problem of differentiating services directly. A final and ideal solution is to restore service differentiation either by developing a method to identify confounding socket traffic or by classifying network traffic in an untrusted network environment some other way. This list of possibilities represents our observations and is by no means exhaustive. More research in this area is required.

6 Future work

This paper establishes the confounding socket as thoroughly resistant to current deterministic classification methods. However, the very randomness that precludes deterministic identifiers may serve as an effective probabilistic one. If confounded traffic is the only type that appears completely random, it can be implicitly identified without computationally expensive decryption. Empirical evaluations are needed to determine if other types of network flows appear random and if evaluation of flow randomness at the network boundary is computationally feasible.

We demonstrated Gnutella as an example network that could be gradually migrated to use the confounding socket. There are several other overlay networks that could benefit from implementing confounding transport. The Freenet project [4] has made peer anonymity a design goal. While the current protocol conceals the file exchange process, our technique offers a method for avoiding real-time ability to determine if a user is even running a Freenet peer. Also, because object exchange is facilitated through the overlay network rather than by the direct connection of two peers, a single peer serving as a gateway node would offer file and routing access to the entire Freenet network. Gnutella gateway nodes expose the entire network's *routing* ability, but allow file exchange only with other peers employing confounding transport.

7 Conclusions

The recent and rapid growth in popularity of peer-to-peer applications has dramatically increased the demand for bandwidth at the network boundaries of large organizations. Rather than attempt to counter this demand by purchasing more resources, many administrators have elected to use QoS devices employing layer 7 protocol analysis to control resource use by deterministically identifying bandwidth intensive protocols.

This paper establishes the confounding socket as a viable and practical means of obscuring protocol structure via encryption by eliminating the connection buildup phase of existing encryption protocols. Our method produces effectively random data, is easily implemented, and scales well against brute force efforts to recover structure. This technique represents a fundamental flaw in existing layer 7 analysis. Future research in statistical and other methods of identification may provide insight into the issues raised here.

In conclusion, the confounding socket invalidates current wisdom regarding flow classification. Deterministic protocol identification is not always possible. Practically, this weakness represents a profound need for new methods of classifying flows as it surely will be exploited by protocols with incentive to circumvent bandwidth controls.

References

[1] E. Adar and B. Huberman. Free riding on gnutella. Technical report, Xerox PARC, 2000.

[2] Lee Breslau, Edward W. Knightly, Scott Shenker, Ion Stoica, and Hui Zhang. Endpoint admission control: architectural issues and performance. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 57–69. ACM Press, 2000.

[3] Cisco. Network-based application recognition. `http://www.cisco.com/warp/public/732/Tech/qos/nbar/`.

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[5] Cornell's commodity internet usage statistics. Available in `http://www.cit.cornell.edu/computer/students/bandwidth/charts.html`.

[6] T. Dierks and C. Allen. RFC 2246: The TLS protocol version 1, January 1999.

[7] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking (TON)*, 7(4):458–472, 1999.

[8] Gnutella. `http://gnutella.wego.com/`.

[9] Tiffany Kary. Cable companies move to tiered pricing, 2002. News.com. Available in `http://news.com.com/2100-1033-885299.html`.

[10] Kazaa. `http://www.kazaa.com/`.

[11] Limewire. `http://www.limewire.org/`.

[12] Holger Luemkemann. Gnutellasim. Available in `http://gnutellasim.limewire.org/`.

[13] G. Marsaglia. Diehard, 1996. `http://stat.fsu.edu/~geo/diehard.html`.

[14] Morpheus. `http://www.musiccity.com/`.

[15] Napster. `http://www.napster.com/`.

[16] Sitara Networks. Qosworks. `http://www.sitaranetworks.com/products/QoSWorks/`.

[17] Packeteer. Packetshaper. `http://www.packeteer.com/products/packetshaper/`.

[18] D. Plonka. Uw-madison napster traffic measurement, Mar 2000. Available in `http://net.doit.wisc.edu/data/Napster/`.

[19] Jordan Ritter. Why gnutella cant scale. no, really, 2001. In *Preprint* `http://www.darkridge.com/~jpr5/doc/gnutella.html`.

[20] Scott Shenker. Fundamental design issues for the future internet. *IEEE Journal on Selected Areas in Communication*, 13(7), September 1995.

A Implementation

Implementing confounding transfer is fairly straightforward and best described using the following pseudocode. Data transfer methods are described below. Connection buildup is described in Figure 1.

Listing 1: Data output method

```
write(Buffer inputData, int length) {
    Buffer traffic

    append length to traffic
    append random integer to traffic

    encrypt first 8 bytes of traffic

    append random integer to traffic
    append input data to traffic
    encrypt traffic following header

    int breakPoint = random integer
        between 0 and traffic size

    write first breakPoint bytes of
        traffic to network
    flush network output buffer

    write remaining bytes of traffic
        to network
    flush network output buffer
}
```

Because incoming data must be decrypted in blocks, we must maintain an input buffer which is filled whenever necessary.

Listing 2: Data input method

```
read(int n) {
    Buffer inputData
    while(internal buffer size < n)
        call fillBuffer

    inputData = extract n bytes from
        internal buffer from buffer
        start

    return inputData
}
```

Listing 3: Buffer management method

```
fillBuffer() {
    int dataRead
    Buffer sizingBytes, messageBytes

    read 8 bytes from network
        to sizingBytes

    decrypt all of sizingBytes

    int dataSize = first 4 bytes of
        sizingBytes

    read dataSize + 4 bytes from
        network to messageBytes

    decrypt all of messageBytes

    add messageBytes from offset
        4 to the internal buffer
}
```