Rook: Using Video Games as a Low-Bandwidth Censorship Resistant Communication Platform

Paul Vines
University of Washington

Tadayoshi Kohno
University of Washington

Tech Report: UW-CSE-2015-03-03
March 20, 2015

Abstract

Censorship and surveillence is increasing in scale, sophistication, and prevalence across the globe. While most censorship circumvention systems are still focused on escaping a given censored region to access Internet content outside of its control, we address a different but equally pressing problem: secure and secret chat within a censored region.

We present Rook as a censorship and surveillence resistant platform for communicaton using online games as its cover application. The use of online games represents a novel form of cover application that provides several features that make them uniquely well-suited for this purpose. Rook transmits data secretly by embedding it in the network traffic of an online game. To defeat current attacks based on deep-packet inspection and traffic shape analysis, Rook does not generate any additional packets, does not change the length of existing packets, and ensures packets that are altered are still valid game packets.

For evaluation, we implement Rook using the online first-person shooter Team Fortress 2. Rook is evaluated against both active and passive attacks demonstrated in recent years including anti-mimicry probes, deep-packet inspection, traffic shape analysis, statistical analyses of packet payloads and game-specific analyses.

1 Introduction

There are increasing concerns about the privacy of online communications throughout the world: there are several countries which have historically acted to both censor and surveil private communications within their borders [4, 29]. However, both surveillance and censorship of the Internet has continued to increase in scale, sophistication, and prevalence [9, 15, 33]. There have been many systems developed for attempting to hide communications on the Internet from censors we will simply refer to these as circumvention systems. Most of these are designed with the intent of routing them outside of a censored region, These systems have many different approaches and goals with respect to circumventing censorship, evading surveillance, and what kinds of attacks they can withstand. With the development of circumvention systems there has been an accompanying development in attack techniques for detecting or blocking them. This has led to an arms race situation in which circumvention systems are invented and improved followed closely by attacks against them similarly being invented and improved.

The focus of most work in reaction to these developments in censorship remains on enabling users to access censored websites by routing their traffic outside of their state's region. We focus on a related but different problem: enabling uncensored and unsurveilled communication between parties within a region of censorship. Recent studies have shown that chat clients with peer-to-peer capabilities are being censored [5] and there is no fundamental reason why a censor cannot block chat programs that use strong security or are unwilling to cooperate in censorship.

In this paper we present Rook as a low bandwidth low latency (approx. 30 bits/second) censorship resistant communication platform. While Rook can be used to secretly communicate any kind of data, the structure of its network and limitations of its bandwidth make it best suited for chat applications. We intend Rook to be used to facilitate secret IRC-like services among users. These services can run the Off-The-Record (OTR) protocol [2] between clients to ensure end-to-end security even from the Rook server. Rook was partially inspired by recent works on pushing circumvention systems further into the application layer [6, 13], but also represents a new direction for circumvention systems in several ways: first, it utilizes the normal network traffic of online games to hide its data; to our knowledge this is first system to leverage this type of application. Games have a number of features that make them ideal host applications for hiding data which are further explained below. Second, Rook alters the host game traffic in full compliance with the application's protocol; this means an adversary must first commit the resources to develop a program to correctly parse the application protocol, and then must also commit computational resources to each suspected Rook connection since the adversary cannot use a stateless single-packet inspection model to detect a Rook connection. As is the case with all of these circumvention systems, Rook is not necessarily undetectable, and we outline possible new research directions for attacks below (see Section 5 and 6). However, Rook does represent a significant increase in the cost of trying to detect and block its use.

Many different types of applications have been used as cover or host applications for previous circumvention systems. Skype in particular has proven popular because its calls operate on a peer-to-peer connection and are assumed to be a reasonably legitimate activity for a user to be engaged in [13, 21]. Multiple TCP/IP-header based covert channels have also been developed. Online games provide a similar opportunity but with greater deniability on the part of users. Traditionally, many online games have allowed individuals to host their own private servers to reduce the resource burden on the companies making the games. This is particularly the case for the genre of game Rook is primarily focused on: the First Person Shooter (FPS). The existence of privately-hosted servers creates opportunities for communities to arise and causes many regular players of these games to play almost exclusively on one or a handful of servers. This provides a completely legitimate cover for a Rook user to repeatedly connect to the same game server over and over to communicate with the other Rook users also connecting to these game servers. Furthermore, legitimate players will often play for hours at a time, day after day. This could be significantly more suspicious in the case of another application, such as Skype, repeatedly being used to contact the same IP for hours at a time every day. Finally, like VoIP services, games are a widespread and popular form of network use [26]; we believe a censor would face similar dissent to a decision to block all Internet-gaming as they would to blocking all VoIP. The general design of Rook is not specific to a game, and so if a censor attempted to block Rook by blocking a single game, it could be adapted to another.

Another advantage of games is that they do not imply actual communication: a Skype connection inherently implies the two parties are sharing information, while a game client connecting to a server does not imply any more communication than the data packets being used to play the game. Since the connection is not encrypted, an adversary could easily detect if there is other information being sent via chat or voice in the game. This creates a plausible deniability that Rook users are connected in any way. Where a Skype connection implies two IPs are exchanging information, a game connection only implies two IPs happen to be playing the same game, and probably are not even aware of who the other IP is. Finally, games provide a way to exchange information while not interrupting the legitimate activity. In most cases the host application, if it is even being run, is not performing its normal operation: e.g., Skype when being used to proxy web traffic is not also providing a true VoIP conversation [13, 21]. Instead, the game is actually being played normally while information is exchanged; the adversary could plant a user in the same game server and not observe any significant difference in how a Rook user played versus a normal legitimate player (see Section 4).

Rook was developed with the FPS as the primary type of game to be used: this is because of the prevalence of private servers and the use a robust and low-latency UDP-based network protocol. The use of UDP allows Rook to modify game packets without creating any additional packets and also not effecting legitimate gameplay. FPS games generally feature between 8 and 128 players on a server at a time. Each player controls one avatar inside a 3-dimensional game world in which they attempt to maneuver and kill the other avatars. As an example implementation, Rook uses the FPS Team Fortress 2 [27], based on the Source Engine from Valve Software. The Rook design can also be used for other types of games, although it may require modification particularly in the case of TCP-based games.

The rest of this paper is laid out as follows: Section 2 describes the design and architecture of the Rook system. Section 3 describes our example implementation of Rook for Team Fortress 2. Section 4 contains our evaluation of Rook against a a variety of current and future proposed attacks including a game-specific trigram value analysis. Section 5 describes related works and how Rook's approach fits into the context of current circumvention system research. Section 6 discusses potential future works in this space building upon Rook. Section 7 contains a summary of Rook's contributions and analysis of our implementation.

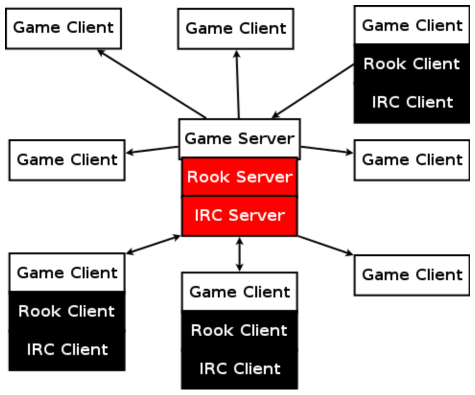

Figure 1: Example formation of a Rook network: one Rook server with multiple Rook clients and normal game clients connected to it.

2 System Design

Rook is a system to facilitate secretly passing data between two Rook clients and a Rook server, operating on machines running game clients and a game server, respectively. In this case, secret is defined as the act of sending the data being unobservable from to an outside observer, as well as the contents of the data itself being encrypted. The data channel between a Rook client and Rook server is composed of two one-way channels to create an overall bidirectional data channel. The creation of the channel requires a shared secret key between the Rook client and Rook server.

2.1 Intended Use Model

While Rook is fundamentally capable of secretly transporting any kind of data, specific features and limitations of both Rook and online games make it particularly suited towards functioning as a secret chat server. Rook provides low bandwidth but also real-time communication, this mostly precludes it from being used for higher-bandwidth tasks such as normal web-browsing or viewing censored videos and pictures. Furthermore, because an online game is being used as the cover application, it would be odd for a user to connect to a server that is in a distant country, since such a server would typically have high latency compared to a more local server.

The intended model for Rook use is a local Rook server running on a machine that is running one or more private game servers used by both Rook users and normal game players. This Rook server would then also run a chat server such as a standard IRC server, that is tunneled through Rook. Rook users who possess a shared key with the Rook server can then group-chat with other Rook users on the same server as they would on a normal IRC server, but without having to fear censorship. For private one-on-one chats Rook users could easily use Off-The-Record messaging over Rook to ensure privacy even from the Rook server. We discuss our prototype implementation which includes these features in Section 4.

2.2 Threat Model

For the rest of the paper we assume the following threat model for our monitor adversary, similar to the standard warden and prisoners threat model used in steganography [1]. The monitor is attempting to detect and/or block the exchange of any information besides legitimate game information. The monitor is considered to have the following capabilities:

- All network traffic between all client and the server are observed
- The normal application traffic is unencrypted
- The monitor can store all data observed over a gaming session and run statistical analyses on it
- The monitor has no internal access to the devices running the game clients and server
- Users can conduct a 1-time secure rendezvous to ex-

change a shared secret key

- The monitor can join the server and observe the actual gameplay (this is not possible if the Rook server is password protected, as some private servers are)
- The monitor does not wish to disrupt legitimate gameplay of innocent users
- The monitor can conduct some active probing of game client and servers, but cannot disrupt legitimate traffic for extended periods of time
- The monitor seeks evidence that information, aside from normal game data, is being exchanged

2.3 Criteria for Success

Rook is a successful censorship resistant platform if the monitor cannot either:

- Positively identify use of Rook more often than falsely identifying legitimate game traffic as use of Rook
- Successfully disrupt Rook communication without impacting legitimate gameplay.

We assume that even if Rook becomes popular, it will still represent a minority of game traffic. Therefore, the first criterion is subject to this difference, so that even a small false positive rate in a detection scheme can yield larger absolute numbers of false positives than true positives. Additionally, the efficiency and practicality of deploying a given detection scheme should be taken into account. General-purpose Deep-Packet Inspection (DPI) techniques, for example, are easier for an adversary to use on a wide-scale than specialized detectors that require storing and analyzing entire traffic captures.

2.4 System Overview

The essence of Rook's scheme for secretly sending data is to identify portions of game packets that can contain many different values and then infrequently select a packet and replace some of that mutable data with other legitimate game values representing the secret data it wishes to communicate. A key insight underlying Rook is that game packets can be modified without affecting gameplay if done carefully, because the game assumes unreliable transport of packets. On the receiving end, the receiver does the inverse: it also identifies these mutable portions of the packet and decodes the values inserted by the sender back into the secret data. This concept is fundamentally simple but grows in complexity in the context of defeating the various kinds of attacks that can be launched against it. The following sections explain in detail the components of the Rook sending and receiving scheme and why they are designed the way they are. Appendix A contains diagrams illustrating the process of the Rook sender and receiver processes.

2.5 Mutable Fields

Rook relies on finding *mutable fields* within game packets. In theory, we could replace any arbitrary bits in the payload of the packet with the secret data, because the packet will never be sent to the actual game process on the other end. However, many bit values are immutable with respect to the protocol of the particular game being used.

For example, many games have application-level sequence numbers in the payload, and overwriting these with arbitrary data would cause the application to issue an error and/or crash.

A relatively easy attack against Rook would be to passively duplicate game packets as they traverse the network and try to parse them using the game's protocol; if they frequently fail to parse then it is unlikely they are merely being mangled in-transmission and so reasonable to suspect use of Rook. Therefore, the implementation of Rook must correctly implement at least part of the game network protocol in order to be able to parse packets and correctly identify which bits are part of *mutable fields*. These are the only bits which are modified when Rook sends data.

In addition to this requirement, the values of some fields are reflected by subsequent packets sent by the receiver of the value. For example, the value of the weapon switch command sent from the client to the server is reflected by subsequent server-to-client messages confirming which weapon is now selected. A passive attacker could relatively easily observe the original field being sent, and then observe whether or not its value was reflected in subsequent return packets. If the packet containing the original field was dropped this field value would not be reflected and the attacker could suspect the use of Rook. Fortunately there are relatively few fields like this, and so the packet parser module for Rook must simply be set to never alter packets containing these fields.

2.6 Symbol Tables

Unfortunately even if the *mutable fields* of a game packet can be correctly determined, not all combinations of these bits are necessarily valid or reasonable to exist in normal game traffic.

For example, imagine a *mutable field* sent from the server to the client representing the velocity vector of another avatar in the game. That value might be encoded as a 16-bit float. However, it might be that the velocity of an avatar is never actually greater than 100.0. If Rook replaced all 16-bits with arbitrary data it could easily be spotted by an adversary parsing packets and looking for velocity vectors with values greater than 100.0.

To prevent this type of attack, Rook does not insert the raw bits of the data it is sending. Instead, Rook keeps a *symbol table* for each *mutable field*. This *symbol table* is constructed by observing normal gameplay at the start of a game connection. For each *mutable field* encountered in the observed data a count of what values it had is kept. This count is then translated into a frequency of a certain value appearing in the *mutable field*. The *symbol table* for that *mutable field* is generated by first pruning values whose frequencies are more than two orders of magnitude less than median frequency. Less frequent elements of the pruned list are then removed until the length of the list is a power of two; that list is the *symbol table* for that *mutable field*. To our knowledge, this is a unique approach for disguising data in a circumvention system; we believe similar approaches could be adopted by other circumvention systems to improve their undetectability.

When Rook attempts to send data by altering a *mutable field*, rather than writing k-bits of its secret data to the field, it instead converts n-bits of secret data into a symbol using the *symbol table* for that *mutable field*, where $n = log_2(length\ of\ symbol\ table) <= k$. By using a *symbol table* generated from normal game traffic Rook avoids sending values that are never or very infrequently seen and so prevents an adversary from filtering traffic based on these.

When the altered packet is received by the other Rook user, the process is reversed to translate from symbols in *mutable fields* to secret data bits, using the same *symbol table* in reverse. Because of this, it is required that both sides possess a copy of the same *symbol table*.

In earlier iterations of Rook a static symbol table was used, created from a long gameplay packet capture. However, the n-gram analysis described in Section 4 showed a flaw in this scheme: the values of some *mutable fields* are dependent upon client configurations (such as graphics settings) and thus created anomalous values in Rook datasets when Rook was used under different settings because the values in the *symbol table* never appeared naturally. Instead, Rook now generates a *symbol table* dynamically when the client first connects to the server. This way, all values sent using Rook represent values not just possible for the game, but already naturally occurring between client and server in a given session. Althogh not shown to be necessary by our analyses, we may add dynamically updating symbol tables to future versions of Rook, as discussed in Section 6.

2.7 Dropping Altered Packets

When Rook receives an altered packet it extracts information as described above, but it also drops the packet in order to minimize the effect running Rook has on the game. Even though all packets altered by Rook are valid game packets, their effect could still potentially cause other players in the game to notice something odd occurring.

For example, if there was a message from the client to the server relaying a command to turn right, this message could be overwritten by Rook to instead turn left. Depending on the game context, a change like this might be unnoticeable or might seem strange. The network protocols of the types of games Rook is designed to use are typically robust to packet drops, so simply dropping the altered packet has less impact on the game experience for both the Rook user and other players than letting the packet reach the game client or server.

2.8 Scheduling Altered Packets

Even though these network protocols are designed to tolerate poor network conditions, gameplay does degrade if too many packets are dropped in a row. Therefore, Rook is designed to let most packets through; by default it selects roughly one-in-ten packets to be altered. Since the values inserted by Rook are only legitimate game values, we cannot send a short flag indicating which packet is the one that has been altered, because that flag might naturally occur without the sender having written it. Any flag long enough to be satisfyingly unlikely to occur randomly is too much overhead for the bandwidth of Rook or would be easily detectable by an attacker. Therefore the Rook sender and Rook receiver must pre-schedule which packets will be altered.

Before this schedule is arranged, the sender and receiver initially synchronize using a flag value. This flag value is derived from their shared key to prevent an attacker from either easily enumerating all Rook servers by sending them flags, or passively detecting Rook channels by looking for the initial flag value being sent. The sender picks an arbitrary packet with enough *mutable fields* to store the flag, default of 40-bits in length. The sender then synchronizes their sending schedule based on the sequence number of the packet they put the initial flag in. The receiver scans received packets until they see this flag; when they receive the flag the receiver also synchronizes its receiving schedule based on that packet's sequence number. In this way, both the sender and receiver have a synchronized schedule for which packets will be altered by Rook. In our implementation, the server performs this receiver-side scanning for all clients for the first 5 minutes of their connection and then assumes they are not potential Rook clients until they reconnect again.

2.9 Shared Deterministic Cryptographic Random Number Generator (DCRNG)

This solution by itself is not necessarily sufficient because of the regularity of packets being altered. If the interval between altered packets is a constant value, an attacker could launch statistical attacks to compare the values of sets of packets and would likely easily be able to determine the difference in a sequence of packets that have all been altered versus a sequence of completely unaltered packets.

To avoid this, the sender and receiver use a shared deterministic cryptographic random number generator (DCRNG) with a seed derived from their shared secret key. This DCRNG is used to produce a set of sequence numbers to use as the shared schedule for which packets to alter.

2.10 Shared Keys

As the previous two sections show, the Rook client and server need to share a secret key from which they derive the expected initial flags and DCRNG seeds. This key exchange must be performed out-of-band once before the first Rook connection is made. After the first connection is established, however, the client and server can easily negotiate a new shared key. Furthermore, to ensure forward secrecy, they can share this key via a standard Diffie-Hellman key exchange. The method used for the initial out-of-band exchange is out of scope of this paper. To further help prevent key compromise from affecting multiple clients, the Rook server is expected to keep a list of shared keys, and distribute a different key to each Rook client.

2.11 Message-Present Bits

A disadvantage of the scheduled sending scheme is that the receiver must assume the scheduled packet has been altered, and so must assume any values within its *mutable fields* represent secret data. To prevent the receiver from erroneously interpreting these values when the sender actually simply had nothing to send, the Rook system uses a message-present bit in each altered packet. To help avoid detection the *mutable field* this message-present bit is inserted into is randomly determined using the same synchronized DCRNG described above.

When a sender inserts data into a packet, they insert a 1 for the value of the message-present bit, translated into a value using the *symbol table* in the same way all other data is, indicating data is present. If the sender has no data to insert, they insert a 0 for the message-present bit, and no additional data.

When the receiver attempts to extract hidden data from the packet, they first check the *mutable field* containing the message-present bit. If it is a 1 then they receive the rest normally, if it is a 0 then the receiver stops there and does not attempt to retrieve any hidden data.

2.12 Packet Loss Resilience

The intention of Rook is to serve as a reliable communication channel. However, it is using a UDP network protocol built to be robust to missing data rather than retransmitting lost packets. Therefore, Rook must include its own layer of reliable delivery to prevent either naturally poor network conditions or an active adversary from easily disrupting the communications.

Rook uses a simple sequence-number/sequence-number-ack header prepended to the message data. Because all the game protocols so-far observed contain an incrementing sequence-number in the payload of

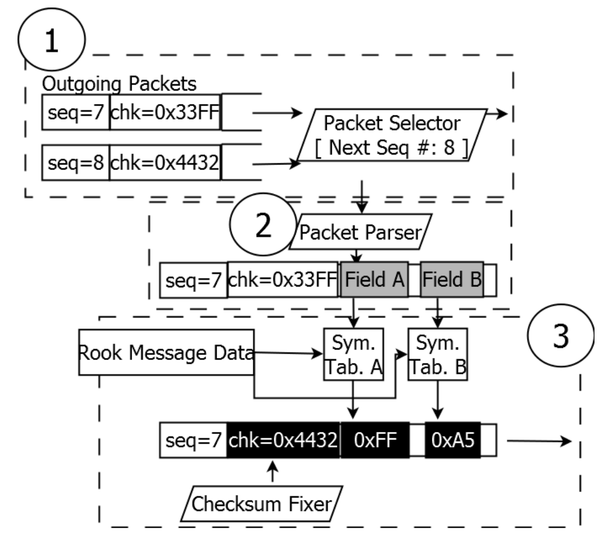

Figure 2: Rook Sending Process: (1) Rook selects a packet to alter based on sequence #. (2) Rook parses the selected packet to find *mutable fields*. (3) Rook overwrites *mutable field* values based on the data to be sent and the *symbol table* for that *field*. Then the checksum is recomputed and the packet sent.

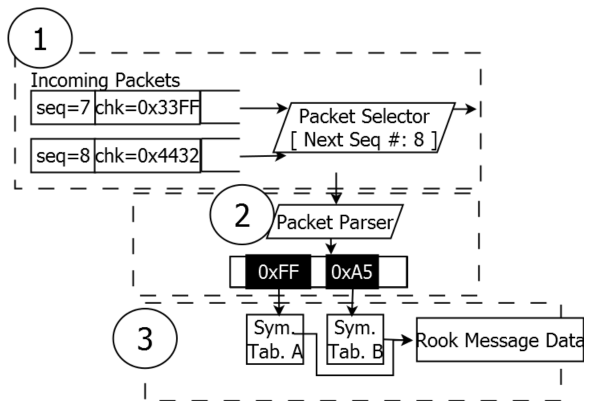

Figure 3: Rook Receiving Process: (1) Rook selects the altered packet based on sequence #. (2) Rook parses the selected packet to find which bits are in *mutable fields*. (3) Rook uses the *symbol tables* corresponding to the *fields* to convert their values back into bits of data. The altered packet is then dropped before reaching the game.

the packet, the Rook receiver can easily determine if a game packet containing Rook data was dropped. In this case, the receiver simply does not increment its sequence-number-ack value and the original sender will understand it needs to retransmit. This is fundamentally the same mechanism used for reliable delivery in TCP, but Rook takes advantage of the game already providing sequence numbers to avoid that overhead.

Keeping the DCRNG synchronized through a packet loss event is an additional complexity. The solution is for the Rook system to always generate a fixed number of random numbers per-packet to use for packet processing (such as determining which *mutable field* is the message-present). When the receiver observes a packet drop based on the missing game packet sequence number, it simply generates the fixed number of random numbers just as if it had received the dropped packet. While the packet data is lost, the DCRNGs of the sender and receiver stay synchronized so the data can be retransmitted. In this way, Rook can be robust against packet-dropping attacks that are not large enought to cause the game connection to be broken (typically 30 seconds or less).

2.13 Client-Server Connection Setup

A Rook connection consists of two active channels connecting a Rook client and Rook server to allow data to covertly flow in either direction. The Rook system does not assume these channels are established as soon as the game connection is established, a Rook server also does not have any ability to recognize a Rook client except by data sent over the Rook channel. Therefore, a special connection handshake is needed to establish the two Rook channels to connect the Rook client and Rook server. The Rook server contains a list of secret keys, S, that are used as the shared keys to facilitate the covert channel. A given Rook client only uses one secret key, S_i to connect to a given server. For simplicity, the following description of the connection handshake is described as if the Rook server also only had a single key, S_i.

Initialization Both the Rook server and client use their shared secret key S_i to seed their deterministic random number generators ($DCRNG_S$). The $DCRNG_S$ is then used to generate a pair of flag values, (F_C, F_S), and two lists of indices, $(I_{C,0}, ..., I_{C,n}, I_{S,0}, ..., I_{S,n})$. The client and server also each generate an AES key from output of the

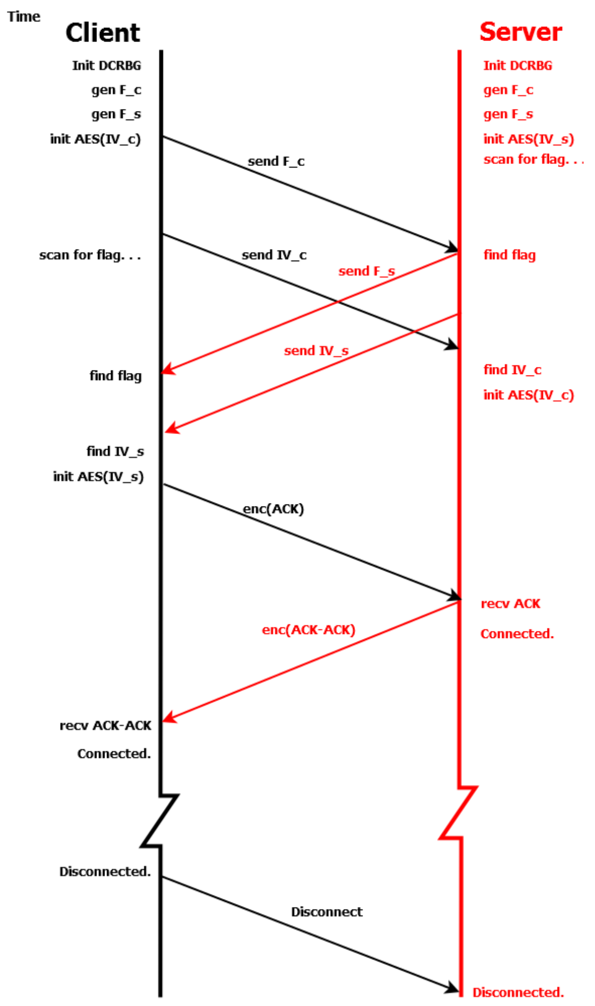

Figure 4: Rook Connection Handshake

$DCRNG_S$ and store the IVs to transmit later.

The server inspects packets from each game client that is not already a Rook client. It performs essentially the same procedure as a receiver, except there are *message-present bits*, and it only checks specific *mutable fields* based on pseudorandom indices derived from the shared key (see above).

Client Sends Flag When the Rook client wishes to connect, they begin filtering outgoing packets to find one with enough free bits to fit the flag value F_C. They then embed F_C into the payload using a slightly modified version of the sending algorithm above: there are no *message-present bits*. The client then uses this packet's application-level sequence number to compute its *next_packet* value to know when to send next, and begins sending its IV.

The client also begins listening for F_S in incoming packets as described above.

Server Finds Flag When the flag value F_C is detected the Rook server immediately calculates the *next_packet* of its receiver based on this packet's application-level sequence number, just as the Rook client did when it embedded the flag. The client sender and server receiver are now synchronized, because their *next_packet* values are the same and their $DCRNG_S$ are synchronized to output the same random values. The Rook server now begins receiving the client IV.

Additionally, when the flag is received, the Rook server begins attempting to embed F_S into outgoing packets in the same manner the client embedded F_C into its outgoing packets (described above), followed by the server's IV.

Server Receives IV When the server receives the client's IV, the server initializes its decryptor with the shared key and client's IV. The server then waits to receive an encrypted "ACK" message from the client. When this is received, it considers both Rook channels to be working and the connection established, and sends its own encrypted "ACK-ACK" message to the client

Client Receives Flag and IV When the client detects F_S it synchronizes its receiver by generating the *next_packet* value. As before when the server received F_C, upon receiving F_S the client receiver and server sender are now synchronized.

The client now initializes its decryptor with the server's IV and shared secret key from subsequent messages. When it has received the entire IV, it generates the a decryptor for the server's messages. Furthermore, the client also encrypts an "ACK" message and sends it to the server.

Client Receives ACK After sending its encrypted "ACK", the client waits for the server to send back an encrypted "ACK-ACK". When this is received the client considers both Rook channels to be working and the connection to be established.

2.14 Client-Client Connection Setup

At this point, the client is connected to the server, but is not communicating with other clients. Fundamentally, these connections simply consist of the Rook server forwarding data between the two clients. Depending on the preferences of the server, it could list all clients connected to it and allow them to establish conversations with each other (as with IRC), or it could hide this information and force clients to already know something like a username or a secret key to try to message a particular client. When two clients establish connections to one another they have the option to establish an Off-The-Record conversation to ensure confidentiality from the Rook server.

3 Implementation

The preceding section described the design of the Rook system, which could be applied to any games with certain network characteristics. To study and evaluate the effectiveness of this system we implemented it for the game Team Fortress 2. Because Team Fortress 2 is built on the Source Engine, it shares the same network stack as other popular Source Engine games including Counter Strike: Global Offensive, and Day of Defeat: Source. Therefore, our implementation can also function for these, although the symbol tables would need to be regenerated to assure they still contain reasonable values for the specific game. All the testing and evaluation was done using Team Fortress 2.

The main challenge to implementing Rook for a given game is reconstructing the packet format for the game. Both sides of the Rook connection must be able to parse the packet enough to correctly identify bits that can be altered without causing the packet to provoke errors from the real game packet parser. In practice this means the Rook code needs to be able to successfully parse the most common types of packets and correctly identify any others to be safely ignored. A description of what we determined about the Team Fortress 2 packet structure can be found in Appendix A.

3.1 Basic Packet Structure

We determined the packet structure of a Source Engine packet sent from both the server and client to be as follows:

- 32-bit Sequence Number - increments on each packet sent, used to determine ordering and duplicate packets
- 32-bit Sequence Ack Number - the sequence number of the last packet received
- 8-bits of Flags
- 16-bits of checksum computed over the rest of the packet - uses CRC32 truncated to 16-bits
- 8-bits of reliable state informaton about 8 subchannels
- If one of the flags is set, then 8-bits are read containing information about choked packets
- Series of net-messages, a 6-bit message type followed by *n*-bits of message data

The difference in packets between the client and server are the types of net-messages they send, described below.

3.2 Client Messages

The client mainly sends two types of messages: *tick* and *usercmd*.

3.2.1 Tick

The *tick* message is simply 48 bits of data. It appears to be necessary for keeping the client and server synched correctly and uses an incrementing value so it is not a good *mutable field* to use.

3.2.2 Usercmd

The *usercmd* message is a message of potentially 14 fields containing information about user commands. Each field is preceded by a 1-bit flag indicating whether that field has been provided or not. These fields represent all the in-game actions a player's avatar can take such as switching weapons, turning, and moving in different directions. In all, the messsage can be anywhere from 13 to 359 bits. Multiple of these messages can be found in a single packet, if the user has issued multiple commands since the last packet was sent. The majority of the fields are based on the user actions, and so are good *mutable fields* for Rook to use. Some of these fields end up being poor candidates due to low entropy: the weapon switch field will typically only have three different values possible at any given time.

3.3 Server Messages

The server also mainly sends two types of messages: *tick* (as above) and *PacketEntity*.

3.3.1 PacketEntity

The *PacketEntity* message contains updates for the entities, the other players and moveable-objects, on the server. Each entity has a number of properties depending on its class. Each of these properties may or may not be updated when the entity is updated. Furthermore, each property is parsed differently depending on its type. Rook primarily modifies character facing and position fields as these appear to be the most frequently changing.

4 Evaluation

4.1 Bandwidth and Usability

Our implementation of Rook for Team Fortress 2 currently operates at 34 bits/second from game client to game server, and 26 bits/second from game server to game client. This is relatively low but still useful for the real-time chat messaging that is the target of this system. As part of our evaluation, we also incorporated an open-source implementation of the Off-The-Record [2] chat protocol to communicate with other Rook clients connected to the same server. The main overhead of the OTR protocol is in the initial key exchange, which can take several minutes in the current Rook implementation. However, after the initial connection is made the secure messages between clients are not significantly slower than unencryted Rook messages.

In addition to raw bandwidth and chat service, it is also important for Rook to avoid disrupting gameplay by clients to prevent an adversary from easily detecting Rook users from within the game. Rook was used in the presence of the built-in Valve Anti-Cheat (VAC) system and did not trigger any kind of action. Additionally, test users of the system experienced no significant degradations of gameplay. The presence of the Rook system intercepting packets did not cause any noticeable latency between the game client and server. The only effect that was ever noticeable was failure to switch weapons if the weapon-switch command was sent in a packet altered by Rook. No other commands were significantly impacted because they were generally held for longer than the span of time for a single packet, and thus subsequent packets that were not modified by Rook conveyed the command.

As mentioned in Section 2, an attack based on detecting discrepancies between a predicted game-state and actual game-state because a packet was dropped by the Rook server is theoretically possible. We consider this attack out-of-scope for Rook because it would require the adversary to essentially run its own instance of the game server and client for each connection it wishes to test, which is impractical on a mass scale. Furthermore, we are not sure that this attack is even possible, in either our chosen implementation or in most other games, due to inaccessible state on the server.

4.2 Censorship Resistance

To evaluate the censorship resistance of Rook we classify the types of attacks we consider into five types:

1. Anti-Mimicry
2. Single-Packet Deep-Packet-Inspection
3. Traffic Shape Analysis
4. Statistical Multipacket Deep-Packet-Inspection
5. Game-Specific n-gram Analysis

4.2.1 Anti-Mimicry

Anti-mimicry attacks are defined as attempts to probe and identify Rook servers or clients based on comparing their response to probes to the response of a normal game server or client. It has been previously shown [7, 12] that many anti-surveillance and anti-censorship systems are vulnerable to these types of attacks.

Rook is not vulnerable to this types of attack because it is not mimicking the game client and server but actually running them on either end. In fact, a monitor can even connect to the server and play normally, including observe the Rook users playing. Because of this, the client and server will also respond correctly to probes the monitor could send, such as malformed packets.

In the case of malformed packets, an active attacker could attempt to send malformed packets in place of the packets they believe are being altered by the Rook system. Since these are dropped on either side before reaching the game client or server, a malformed packet would not generate the expected error response. However, the attacker only has a $1/n$ chance, where n is the packet spacing variable (approximately 10 in standard use, although the actual packet selection is randomized), of replacing an altered packet. The rest of the time the attacker will accidentally replace a legitimate game packet. The only errors visible to a network-level observer are those that cause the client to disconnect from the server, therefore the attacker would need to send a malformed packet that would cause a client disconnect. So in $(n-1)/n$ cases, the game client is disconnected because the adversary modified a non-Rook packet. If the user of this client is actually using Rook, we argue both they and the Rook server would notice the attacker's actions and so avoid reconnecting for some time. In all other cases, the attacker will be disconnecting legitimate clients which essentially becomes the same as blocking the playing of this game on the attacker's network altogether. Thus, we argue that while technically possible, implementing this attack would not be compatible with the adversary's goals.

An active attacker could also attempt to use malformed packets to deny Rook service without detecting it. They could replace mutable field values with other valid values to cause the Rook receiver to receive a value that corresponds to different data in the symbol table, or that is not present in the symbol table at all. As above, this attack will only be effective if the attacker is able to alter the Rook packets, which are randomly distributed. Unlike the above attack, this would not cause the game to disconnect due to malformed data, but it would still degrade legitimate player experience since their commands would be being overwritten with random values.

4.2.2 Single-Packet Deep-Packet-Inspection

Single-packet Deep-Packet-Inspection (DPI) is what many censorship regimes appear to currently use to attempt to filter traffic types rather than just blocking IPs or ports [31]. Similarly to anti-mimicry above, single-packet DPI approaches could potentially block shallower forms of obfuscation that only change some values in a packet to look like another type of traffic. If Rook did not correctly parse the game packets, an attacker could simply run a copy of the game network stack to determine if all packets being sent are correctly formatted.

Rook is not vulnerable to these approaches because it is fully compliant with the game's packet specification, and only uses values from packet data-fields that have been previously observed. Therefore, any single-packet DPI approach would never detect packets altered by Rook because they are still correctly formatted and represent reasonable values in the game. If such an approach was capable of detecting a packet altered by Rook, it would also necessarily detect legitimate, unaltered, game packets because a Rook packet essentially is made of correctly pieced together fragments of legitimate packets.

4.3 Statistics-Based Attacks

While more costly to deploy, and hence less likely, we now consider attempting to detect Rook using multipacket statistical analyses to compare normal game traffic to game traffic altered by the use of Rook.

4.3.1 Data Gathering Methodology

To evaluate the traffic shape analysis and the statistical DPI attacks discussed below, we created datasets as follows: a TF2 server with 20 bots is run on one machine on the LAN. This machine also runs the Rook server in the datasets using Rook. A second machine on the LAN runs a TF2 client, connects to the server, and runs the Rook client with a typical or high-bandwidth (HB) configuration in datasets using Rook. The two configurations only differ in how often packets are selected to be altered. The typical configuration alters approximately 1-in-10 game packets. The high-bandwidth configuration alters 1-in-3 game packets. The purpose of these two configurations is to show that our attacks are reasonable by showing they can identify HB-Rook, but that normal Rook use is still undetectable by them. For datasets using Rook the user connects to the TF2 server, both the Rook server and Rook clients observe 600 packets (approximately 60 seconds of gameplay) in each direction and then create their symbol tables. The Rook client then connects to the Rook server and each side sends pseudo-random data at the maximum data-rate to the server. The packets sent and received after the Rook connection is made are captured using Wireshark for analysis as described below. Each capture is approximately 7,000 client-to-server packets, and 6,000 server-to-client packets (about 5 minutes of gameplay/actual Rook use).

We gathered 61 datasets: 20 using Rook in high-bandwidth configuration; 20 in the typical configuration; 20 of normal gameplay; and finally one of normal gameplay that was used as a baseline for investigating the three datasets listed above.

During the course of experimental setup, we observed that many unexpected factors can affect the values sent in the game packet payloads; these appear to include: operating system, graphics hardware, and game window resolution. To give our attacks the best reasonable environment, we attempted to hold as many variables in the experiment stable as possible. In addition to using the same number of bots in all cases, the game player was also the same and already had experience in the game, the class played was the same, the game level played on was the same, and the level was restarted before each sample. These essentially reflect the best circumstances an adversary could be expected to capture traffic under, as there should be as little variance in how the user is playing, the range of values being sent in packets, and the effects of network infrastructure on traffic shape.

The above efforts to control the environment for more consistent analysis contributed to the dataset size, since the generation could not be automated or parallelized. However, this size appears reasonable for evaluation since most of the samples show strong similarities with only a few outliers.

4.3.2 Traffic Shape Analysis

Traffic shape analysis is conducting statistical analyses on the size and timing features of the stream of packet traffic between the client and server. Some previous covert channels have used timing changes to send secret information, and some have simply injected extra bytes into application packets, e.g., [8, 18]. These approaches have the potential to be detected by comparing statistics between known normal traffic and suspicious traffic, e.g., average size of packets or median timing between packets [14].

Rook should not be vulnerable to these approaches because it does not alter the timing or length of game packets to embed its information, it only alters individual data-fields within the packet. Furthermore, using Rook does not cause any additional packets to be sent, or packets to be changed in size, by the game server or client, so the traffic shape should be unaffected.

To evaluate a traffic shape analysis attack, we chose to extract and compute statistics over the overall bandwidth and the spacings of packets. The first should not be impacted by the use of Rook unless dropping some packets results in significant resending of data. However, since the game protocol is built to be robust to poor network conditions, the bandwidth is not increased by the use of Rook (see Figs 5). The amount of time between packets was analyzed by comparing the distribution of packet spacings to the distribution extracted from the baseline traffic capture using the 2-Sample Kolmogorov-Smirnov (KS) test for statistical significance. This is a straightforward strong test for whether two distributions differ significantly and has been used to evaluate circumvention systems like CoCo in the past [11, 13, 14, 16]. The results of this (see Fig. 5) are two-fold. First, the conservative typical Rook use is difficult to distinguish from normal unaltered traffic in terms of bandwidth used and packet timing. The high-bandwidth configuration of Rook, however, is clearly visible in both cases for its change to the server traffic shape. This is because this configuration causes so many server packets to be altered (and thus dropped on the client-side) that the server appears to resend data in large bursts that have significantly different traffic characteristics. This result shows a traffic shape analysis is a valid potential attack, but is not effective when Rook is used as intended with a relatively low-packet alteration rate.

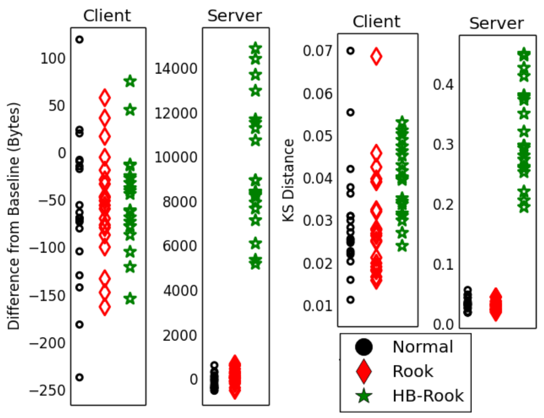

Figure 5: The difference in bandwidth consumed from the baseline traffic and KS-distance between packet timing distributions. HB-Rook is clearly distinguishable in its server bandwidth consumption and timing but Rook is not.

As described above, these data are all from packet captures of a client and server connected on a LAN. In actual use normal gameplay may have larger variation in traffic shape from varying network conditions between the game client and server.

4.3.3 Statistical Multipacket Deep-Packet-Inspection

Statistical multipacket DPI is essentially the monitor taking a packet capture of all the traffic between the client and server and running statistical tests across the payloads

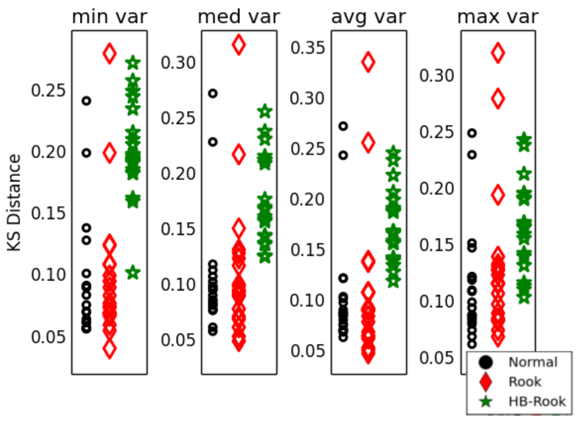

Figure 6: Results of the KS test performed on the distributions of variance across byte positions in client packets of the same size. Average HB-Rook distance is clearly higher than normal, average Rook distance is slightly higher.

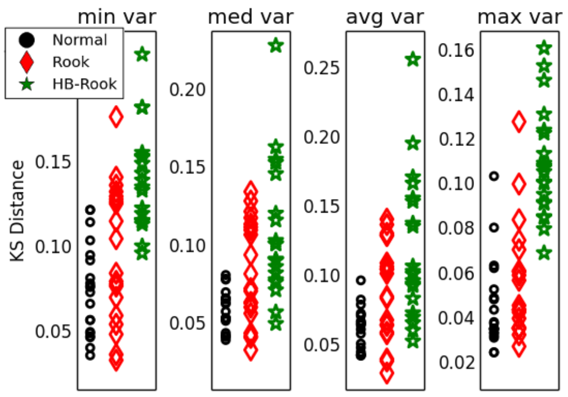

Figure 8: Results of the KS test performed on the distributions of variance across byte positions in server packets of the same size. Average HB-Rook distance is significantly higher than normal, average Rook distance is also somewhat higher.

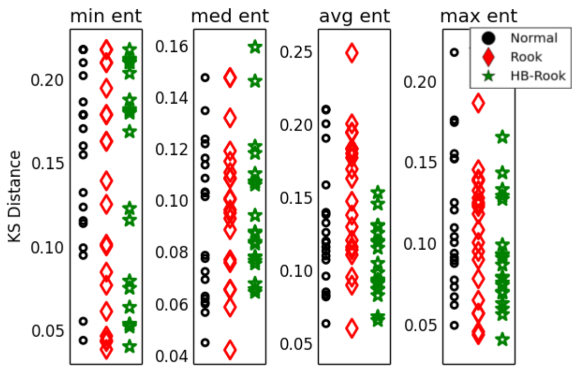

Figure 7: Results of the KS test performed on the distributions of entropy computed across eack packet for client packets of the same size. No feature where Rook or HB-Rook are clearly distinguishable from all normal samples.

and comparing these results to those obtained from doing the same process to traffic from a known normal game. There are many possible tests one could do, and no standard for using this kind of approach to evaluate a channel of Rook's kind. As a starting point, we decided to adapt methods used in the related area of covert-timing-channels to test Rook's detectability. In covert-timing-channel the timing of packets is the information channel and therefore statistical tests are run on the distributions of packet timings (similarly to the traffic shape analysis above) [11]. The analogous channel in Rook is the packet payloads, so we run the same tests on statistics computed over the payloads in place of the timing distributions.

Because we only use values that have previously been observed in normal game traffic, we expect the only potentially detectable difference between the traffic of a normal game and a game using Rook would be in the distributions of these values. Therefore, we chose to measure the variance and entropy across each data-field in a normal game and a game using Rook. We measure the variance by first grouping packets by size, and then computing the variance across all these packets at the same byte position. This process is repeated for each byte position, and for each size of packet captured. Entropy was measured by computing the entropy of each packet for all packets of the same size. The minimum, median, average, and maximum of each of these two statistics for each packet size was extracted to form eight distributions to test. The distributions of all of these results were compared to the same distributions derived from the baseline game traffic,

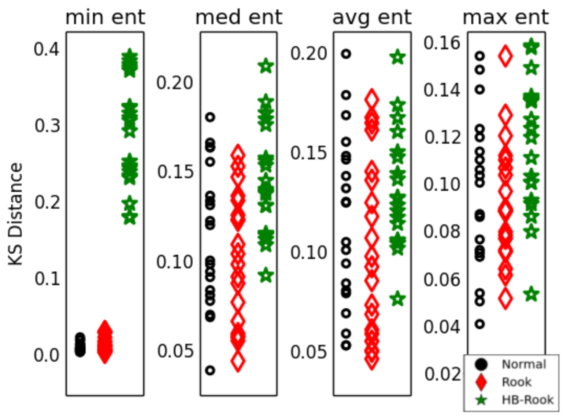

Figure 9: Results of the KS test performed on the distributions of entropy computed across each packet for server packets of the same size. HB-Rook is clearly distinguishable in min ent and med ent, Rook is not distinguishable from normal samples.

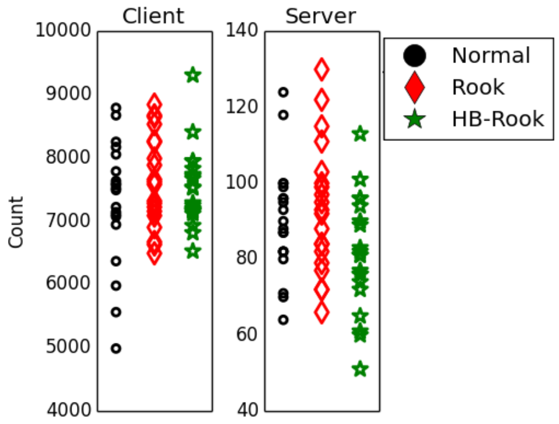

Figure 10: Sample of counts of distinct unigrams for one client field and one server field.

and compared using the 2-Sample KS test.

The resulting graphs can be seen in Figs. 6, 8, 7 and 9. These are each the stated statistic gathered from the sample traffic compared to the same statistic gathered from a baseline normal traffic sample using the 2-sample KS test. As can be seen, a few of these statistics (client variance, server entropy) show significant success against the high-bandwidth version of Rook by how easy it is to separate those data from the rest, showing that they are reasonable attacks to mount against a system like this. However, as with the traffic shape analysis attacks, the normal bandwidth Rook samples are not distinguishable from normal traffic.

4.3.4 Game-Specific *n*-gram Analysis

The final analysis we conducted was a game-specific *n*-gram analysis: this is an analysis of the values observed in the mutable fields of our implementation of Rook for Team Fortress 2. Using the same packet parsing module, we constructed lists of unigrams, bigrams, and trigrams for each mutable field for both the client and server. This analysis is similar to the analysis of variable-bit-rate encoding in VoIP used to distinguish languages spoken without decrypting the datastream [32]. It is important to note that for any of the following attacks, the censor must have a game-specific implementation of data gathering and analysis tools, not general-purpose statistics like those used above. This would require an adversary to build a parsing engine and potentially manually generate a list of the useful features for detecting a specific game. Further, for the actual use the adversary must do a full capture of the traffic and, if blocking is desired, parse and analyze it in relative real-time.

In the following section we show and discuss the results of analyzing solely the unigram and trigram data. We also analyzed the bigram data, but it showed essentially the same results as the trigram data, and so we exclude it here for the sake of brevity.

Count of Distinct Trigrams The first analysis we performed was a comparison of the number of distinct trigrams in each sample. An *n*-gram is distinct if Rook only uses values for mutable fields that have been observed in normal game data, so the number of distinct unigrams is never increased by running Rook. The count of unigrams in both normal and Rook samples can vary, which naturally can lead to more significant variation in bigram and trigram counts (see Fig. 10). To help correct for this, we computed an approximation of the function of count of unigrams to count of bigrams and trigrams based on the normal data samples and then used this function to base-

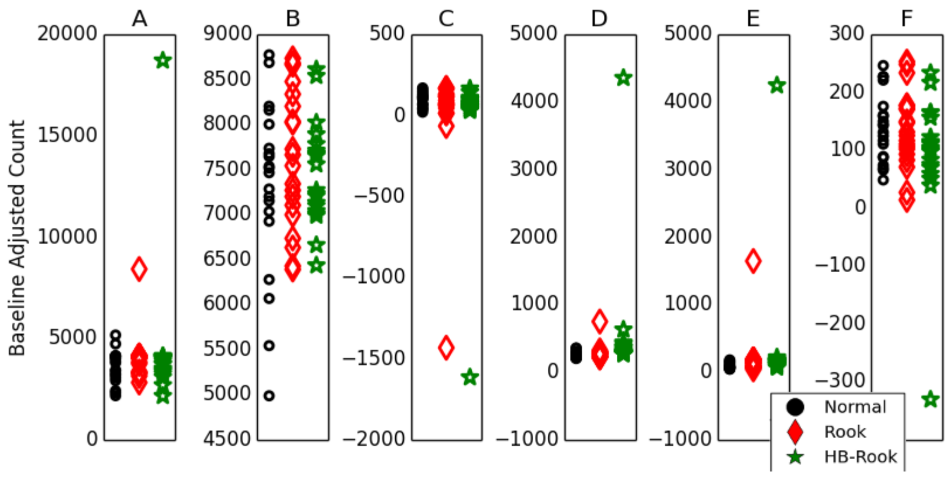

Figure 11: Adjusted counts of distinct trigrams from client traffic. A few outliers for both HB-Rook and Rook.

line each of the sample counts before analyzing. The results in Figs. 11 12 show that there are a few outliers, but most of the normal samples and typical Rook samples are very similar. The high-bandwidth Rook samples, however, are clearly distinguishable in the server trigram counts. We believe this is because the server will be replacing so many mutable fields that would normally contain new values with previously seen values stored in its symbol table. This is significant enough to reduce the number of distinct trigrams by a very noticeable amont. This shows that high-bandwidth Rook use would easily be detected. A detector could potentially be developed to detect the normal Rook traffic, but it would either have a very high false-negative or false-positive rate. If this is still considered practical, we provide mitigating solutions in Section 6.

Frequency Distribution An additional analysis we performed was a comparison of the frequency distribution of trigrams. If Rook is causing unlikely values to occur more frequently, then this should influence the frequency with which trigrams including them appear. Performing a 2-sample KS-test on the frequency distributions of the samples shows a few interesting results (see Figs. 13, 14). For most fields all three sample sets are similar and intermixed, showing no clear way to distinguish even the high-bandwidth configuration of Rook. Interestingly in the angle2-pane of Fig. 13 the high-bandwidth Rook is almost completely distinguishable. However, the KS-Score indicates those samples are actually more similar to the baseline-traffic than either the typical configuration of Rook or the normal gameplay. A similar situation can be seen in the D and E-panes of Fig. 14 where several of the typical Rook samples can be distinguished by being slightly more similar to the baseline than the normal samples. Given the small differences in KS-Score and the overall dissimilarity of all samples, especially in the latter case, this could be a statistical anomaly, as can arise when many statistics are calculated.

Single-Frame Anomalies In the case of client-commands, the client could send repeats of the same command as a result of normal gameplay (for example, holding the forward button to continue moving forward). In trigrams, this would create trigrams of the exact same nonzero value for all three entries. If such a repeating field was a mutable field in our Rook implementation then Rook could inject a different value into the field for a single-frame in the middle of a run of repeating values. This would create what we call a single-frame anomaly, where the first and third values of a trigram are the same value, but the second is different.

We did not intentionally filter these mutable fields from our Rook implementation. However, there does not ap-

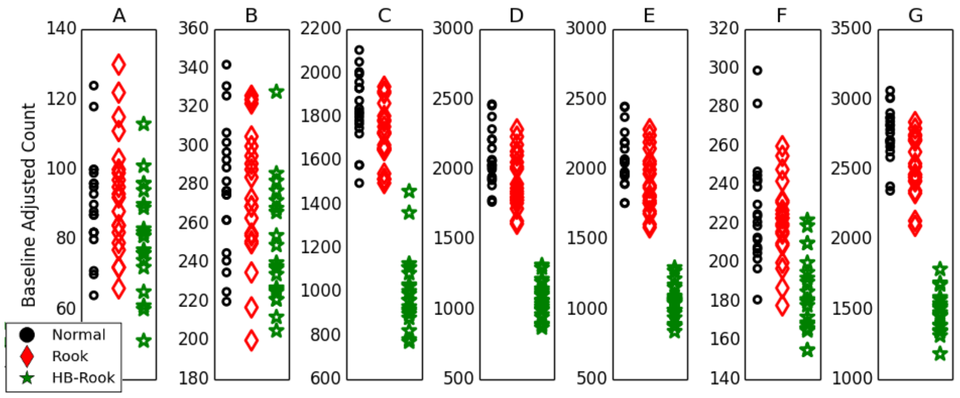

Figure 12: Adjusted counts of distinct trigrams from server traffic. HB-Rook is clearly distinguished over several fields from both normal and Rook samples.

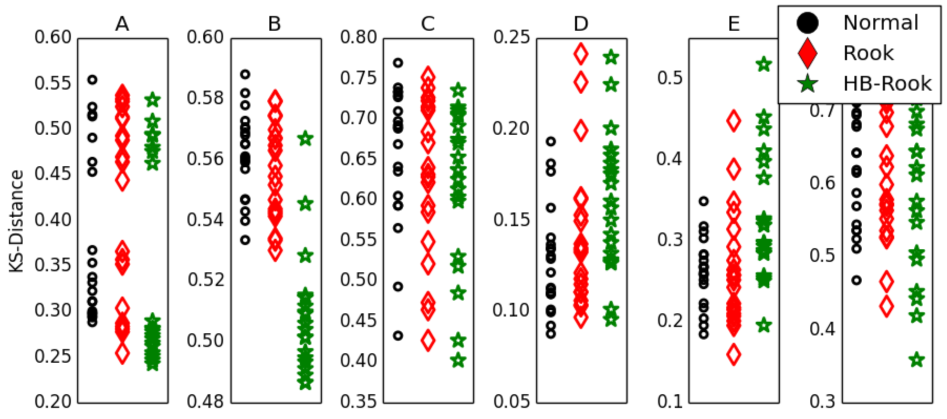

Figure 13: Results of the KS test performed on the frequency distributions of trigrams. One field where HB-Rook is fairly distinguishable but due to being more similar to the baseline sample than normal or Rook samples.

pear to be any consistent difference in number of single-frame anomalies between normal gameplay and Rook use (see Fig. 15). We do not show this analysis for the server's packets because there are no repeating sequences.

In summary, we evaluated the security of Rook against five different major attack vectors. It is robust against standard anti-mimicry, DPI, traffic shape analysis, and generalized statistical attacks. Further, it is resilient against game-specific statistical attacks but may be vulnerable to a focused adversary devoting resources to a detecting use of Rook on a specific game leveraging full packet captures. We believe this represents an improvement in the state-of-the-art in this field and still meets a high level of security since the resources to mount an attack are at the upper bound of our threat model, requiring multipacket statistical analysis using game-specific knowledge.

5 Related Work

There have been many different approaches taken to enabling censorship circumvention or surveillance avoid-

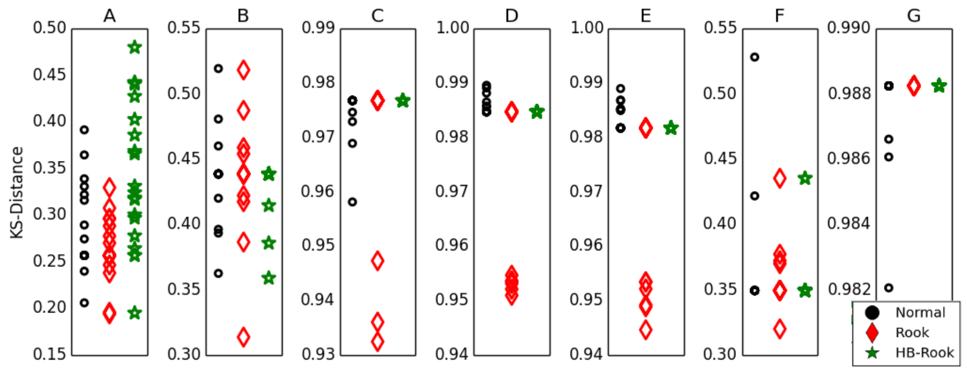

Figure 14: Results of the KS test performed on the frequency distributions of trigrams. Two fields where Rook is distinguishable as being more similar to the baseline than either normal or HB-Rook samples.

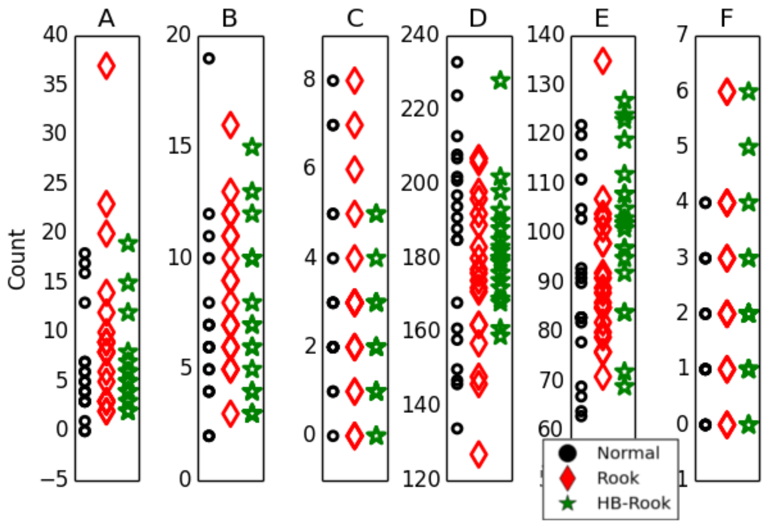

Figure 15: Count of Single-Frame Anomalies in client samples. Showing a few outliers from Rook and HB-Rook but overall not very distinguishable from normal samples.

ance in the past. These have ranged from manipulating traffic shape attributes such as packet timing and size [8, 11], to TCP, RTP, and UDP header bits [20, 22], up to picking specific payloads such as sending specific DNS-lookups to pass information [3]. Recently there have been many systems developed which try to slip past monitors by disguising themselves as normal traffic [13, 19, 23, 30, 34].

Timing Steganography There have been several iterations of systems based on using packet timings to covertly communicate [8, 11]. These systems are potentially harder to detect than Rook because the packet timings they are modifying are already impacted by what other processes are running on the machine, which is outside of the attack scope. However, an active adversary can potentially block the channel by adjusting the timing of packets slightly to destroy the information, or drastically reduce the bandwidth, without necessarily impacting the legitimate application use. Despite these drawbacks, a timing-based covert channel like CoCo could actually work excellently alongside Rook. The bandwidth gain would be modest, about 5-10 bits/second, but the two systems operate orthogonally to one another and so the risk of detection would merely be whichever system is the most detectable.

Header Value Steganography Many fields in standard network protocol headers (TCP, UDP, RTP, etc.) have been found to be useful for steganographic purposes [17, 20, 22]. These include the least-significant-bits of the timestamp, padding values, initial sequence numbers, and various flags such as do-not-fragment. These covert channels have an advantage in that they are ubiquitous to all applications with any standard type of network traffic and so can easily bypass any application filtering an adversary could put in place. However, attacks against the covertness of several of these channels have been shown [24, 25]. Because some of these attributes are not normally used, statistical attacks can detect anomalies from their increased use as covert channels.

Furthermore, these schemes are based on the application ignoring whether these fields are set or not, so an adversary can simply normalize them to deny the covert channel. Rook modifies data that is used by the application, so this type of attack degrades legitimate use (as discussed in Section 4).

Application Protocol Mimicry There have been several recent systems based on transforming the appearance of traffic to evade censorship [21, 28, 30]. These and others have culminated in the development of Tor's Obfsproxy [6] which uses modules called pluggable transports to reform traffic to look like an arbitrary protocol that is believed to be uncensored or unmonitored. These types of approaches generally suffer from weakness to active probing: if either end is not actually running the application the altered traffic would be produced by, probes from adversaries will be ignored. As shown by Houmansadr et al., it is very difficult to try to accurately mimic how a real application will respond to an arbitrary probe, particularly those which would cause errors [12]. Since Rook actually runs the game client and server, an active adversary's probes will be responded to in exactly the same manner as a normal game client and server. Furthermore, depending on the application being mimicked, an adversary could attempt to determine the legitimacy of packets by running them through its own copy of the application. If the mimicry is shallow (to increase bandwidth or reduce the work required to create the channel) then these packets will probably fail this test, providing another way for an adversary to spot mimicry.

Application Subversion Since these attacks on mimicry-based systems have been published, there have been several new systems proposed which hide data at the application-layer, rather than inserting it at the transport layer [10, 13, 19, 23, 34]. These approaches are the most similar to Rook, in that they are essentially sending correct application data. However, they require the adversary to not be able to see what that data is. Unlike Rook, all of these approaches rely on an encrypted channel between the ends of the application. Current events show that some censors will force man-in-the-middle attacks or subverted versions of programs to be used to allow breaking this encryption. Rook does not its applications traffic to be encrypted to remain secure.

Castle On 3/19 another paper was put online that also uses games as covert channels. Their system was named Castle. On 3/20, we released this as a tech report. The two works were concurrent and the similarity of naming is purely coincidental; Castle and Rook use distinctly different mechanisms and types of games to operate and also target different threat models.

6 Discussion and Future Work

Rook is a new approach to an established problem of censorship resistant communication. We argue that online games provide an excellent form of cover for secret communication and have enough mass appeal that a censor would be reluctant to outright block their traffic. The evaluation of our implementation of Rook for Team Fortress 2 shows it to be robust to all currently known forms of network interference and censorship. Further, the evaluation shows Rook is resistant to potential new forms of censorship, such as deep-packet statistical analyses and application-specific analyses, that may become common tools for censors in the future.

Rook also aims itself at somewhat under-represented facets of censorship resistance: establishing safe communication entirely within a censor's region of control, and emphasizing keeping plausible deniability for its users. The current implementation of Rook is functional in exchanging message undetected; however, we believe it could be expanded upon in the following ways:

The first and most obvious extensions from a utilitarian perspective is the implementation of Rook for more games. We developed the Rook code in a modular fashion so that new modules for interpreting different game packet protocols can be easily added.

From a system design perspective, a secure bootstrapping mechanism for finding and contacting Rook servers would be a significant addition to the usability of the system, and could potentially also be conducted over the same online game, but perhaps with a higher-latency mechanism. However, this is a major challenge in circumvention systems in general that has not been solved.

Another improvement that could be made to Rook is making the symbol tables dynamically self-adjusting to attempt to better preserve the traffic statistics. Essentially each side would monitor how the packets it altered have impacted a set of statistics and then modify their symbol tables to try to minimize the statistical deviance created by hiding data. This would inevitably reduce bandwidth to some extent; however, if statistical attacks are a valid and massively deployed form of detection for Rook, the tradeoff would be worthwhile to defeat such attacks.

In the space of attacks, Rook prompts several ideas of more advanced attacks potentially applicable to both itself and other application-layer circumvention systems. Markov-modeling based attacks are potential detectors of Rook use. The attacker would generate a markov-model of user behavior based on parsing large normal traffic captures for the game. The attacker would then try to detect Rook traffic by looking for play that is particularly different from the model. However, there may not exist a model that fits all normal players well enough while still showing Rook as significantly different.

7 Conclusion

In this paper we presented Rook, a system designed to provide low bandwidth low latency censorship resistant communication using the network traffic of online games. Rook represents the first censorship circumvention system to utilize online games as a cover for secret communicaiton. Beyond its novelty, Rook represents a useful addition to the space of existing circumvention techniques and systems. Unlike many other systems, Rook focuses on providing secret communication within a censor's region of control, and presents greater secrecy and deniability than previous systems by virtue both of how its communication is hidden and by it being hidden in online game traffic instead of other applications that would show more differences between legitimate users and censorship circumventing users.

Our implementation of Rook shows that it lives up to its goal of providing bandwidth high enough for chat, including over the OTR protocol, while remaining undetectable using any mass attack methods currently known to be employed by censorship regimes. Furthermore, Rook presents a fundamentally greater challenge to detect than what most current circumvention systems present. An adversary would have to commit resources both to develop an attack against a particular implementation of Rook, and allocate computational resources to each game connection they find suspicious in order to defeat Rook. Even under targeted attack, we argue novel attack techniques would need to be developed to definitively detect Rook communications.

We intend to release our code along with developer documentation to help others interested in censorship resistant communication extend Rook to operate with many more games.

8 Acknowledgements

This work was supported in part by NSF Award CNS-0846065 and by the Short-Dooley Endowed Career Development Professorship.

References

[1] ANDERSON, R. J., AND PETITCOLAS, F. A. On the limits of steganography. *Selected Areas in Communications, IEEE Journal on 16*, 4 (1998), 474–481.

[2] BORISOV, N., GOLDBERG, I., AND BREWER, E. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM workshop on Privacy in the electronic society* (2004), ACM, pp. 77–84.

[3] BOWES, R. Dns cat.

[4] CLAYTON, R., MURDOCH, S. J., AND WATSON, R. N. Ignoring the great firewall of china. In *Privacy Enhancing Technologies* (2006), Springer, pp. 20–35.

[5] CRANDALL, J., CRETE-NISHIHATA, M., KNOCKEL, J., MCKUNE, S., SENFT, A., TSENG, D., AND WISEMAN, G. Chat pro-

gram censorship and surveillance in china: Tracking tom-skype and sina uc. *First Monday 18*, 7 (2013).

[6] DINGLEDINE, R. Obfsproxy.

[7] GEDDES, J., SCHUCHARD, M., AND HOPPER, N. Cover your acks: pitfalls of covert channel censorship circumvention. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 361–372.

[8] GIFFIN, J., GREENSTADT, R., LITWACK, P., AND TIBBETTS, R. Covert messaging through tcp timestamps. In *Privacy Enhancing Technologies* (2003), Springer, pp. 194–208.

[9] GREENWALD, G., BALL, J., AND RUSHE, D. Nsa prism program taps in to user data of apple, google and others. *The Guardian* (June 2013).

[10] HAHN, B., NITHYANAND, R., GILL, P., AND JOHNSON, R. Games without frontiers: Investigating video games as a covert channel. *arXiv preprint arXiv:1503.05904* (2015).

[11] HOUMANSADR, A., AND BORISOV, N. Coco: coding-based covert timing channels for network flows. In *Information Hiding* (2011), Springer, pp. 314–328.

[12] HOUMANSADR, A., BRUBAKER, C., AND SHMATIKOV, V. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on* (2013), IEEE, pp. 65–79.

[13] HOUMANSADR, A., RIEDL, T., BORISOV, N., AND SINGER, A. I want my voice to be heard: Ip over voice-over-ip for unobservable censorship circumvention. In *The 20th Annual Network and Distributed System Security Symposium (NDSS)* (2013).

[14] KHATTAK, S., SIMON, L., AND MURDOCH, S. J. Systemization of pluggable transports for censorship resistance. *arXiv preprint arXiv:1412.7448* (2014).

[15] KILLOCK, J. Sleepwalking into censorship. *Open Rights Group* (July 2013).

[16] LI, S., SCHLIEP, M., AND HOPPER, N. Facet: Streaming over videoconferencing for censorship circumvention. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society* (2014), ACM, pp. 163–172.

[17] LLAMAS, D., ALLISON, C., AND MILLER, A. Covert channels in internet protocols: A survey. In *Proceedings of the 6th Annual Postgraduate Symposium about the Convergence of Telecommunications, Networking and Broadcasting, PGNET* (2005), vol. 2005.

[18] LUCENA, N. B., PEASE, J., YADOLLAHPOUR, P., AND CHAPIN, S. J. Syntax and semantics-preserving application-layer protocol steganography. In *Information Hiding* (2005), Springer, pp. 164–179.

[19] LV, J., ZHANG, T., LI, Z., AND CHENG, X. Pacom: Parasitic anonymous communication in the bittorrent network. *Computer Networks* (2014).

[20] MAZURCZYK, W., AND SZCZYPIORSKI, K. Steganography of voip streams. In *On the Move to Meaningful Internet Systems: OTM 2008*. Springer, 2008, pp. 1001–1018.

[21] MOHAJERI MOGHADDAM, H., LI, B., DERAKHSHANI, M., AND GOLDBERG, I. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 97–108.

[22] MURDOCH, S. J., AND LEWIS, S. Embedding covert channels into tcp/ip. In *Information Hiding* (2005), Springer, pp. 247–261.

[23] RAGNARSSON, B., AND WESTEIN, P. Using git to circumvent censorship of access to the tor network.

[24] SOHN, T., MOON, J., LEE, S., LEE, D. H., AND LIM, J. Covert channel detection in the icmp payload using support vector machine. In *Computer and Information Sciences-ISCIS 2003*. Springer, 2003, pp. 828–835.

[25] SOHN, T., SEO, J., AND MOON, J. A study on the covert channel detection of tcp/ip header using support vector machine. In *Information and Communications Security*. Springer, 2003, pp. 313–324.

[26] VALVE. Steam and game stats.

[27] VALVE. Team fortress 2.

[28] WANG, Q., GONG, X., NGUYEN, G. T., HOUMANSADR, A., AND BORISOV, N. Censorspoofer: asymmetric communication using ip spoofing for censorship-resistant web browsing. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 121–132.

[29] WARF, B., AND VINCENT, P. Multiple geographies of the arab internet. *Area 39*, 1 (2007), 83–96.

[30] WEINBERG, Z., WANG, J., YEGNESWARAN, V., BRIESEMEISTER, L., CHEUNG, S., WANG, F., AND BONEH, D. Stegotorus: a camouflage proxy for the tor anonymity system. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 109–120.

[31] WINTER, P., AND LINDSKOG, S. How china is blocking tor. *arXiv preprint arXiv:1204.0447* (2012).

[32] WRIGHT, C. V., BALLARD, L., MONROSE, F., AND MASSON, G. M. Language identification of encrypted voip traffic: Alejandra y roberto or alice and bob? In *USENIX Security* (2007), vol. 3, p. 3.

[33] XU, X., MAO, Z. M., AND HALDERMAN, J. A. Internet censorship in china: Where does the filtering occur? In *Passive and Active Measurement* (2011), Springer, pp. 133–142.

[34] ZHOU, W., HOUMANSADR, A., CAESAR, M., AND BORISOV, N. Sweet: Serving the web by exploiting email tunnels. *HotPETS. Springer* (2013).