TELEPATH: A Minecraft-based Covert Communication System

Zhen Sun
Cornell Tech
zs352@cornell.edu

Vitaly Shmatikov
Cornell Tech
shmat@cs.cornell.edu

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179335

***Abstract*—Covert, censorship-resistant communication in the presence of nation-state adversaries requires unobservable channels whose operation is difficult to detect via network-traffic analysis. Traffic substitution, i.e., replacing data transmitted by a "cover" application with covert content, takes advantage of already-existing encrypted channels to produce traffic that is statistically indistinguishable from the traffic of the cover application and thus difficult to censor.**

Online games are a promising platform for building circumvention channels due to their popularity in many censored regions. We show, however, that previously proposed traffic substitution methods cannot be directly applied to games. Their traces, even if statistically similar to game traces, may violate game-specific invariants and are thus easy to detect because they could not have been generated by an actual gameplay.

We explain how to identify *non-disruptive* content whose substitution does not result in client-server inconsistencies and use these ideas to design and implement TELEPATH, a covert communication system that uses Minecraft as the platform. TELEPATH takes advantage of (1) Minecraft's encrypted client-server channel, (2) decentralized architecture that enables individual users to run their own servers, and (3) popularity of "mods" that add functionality to Minecraft clients and servers. TELEPATH runs a Minecraft game but substitutes non-disruptive in-game messages with covert content, without changing the game's interaction with the network manager.

We measure performance of TELEPATH for Web browsing and audio streaming, and show that network traffic generated by TELEPATH resists statistical traffic analysis that aims to distinguish it from popular Minecraft bots.

1. Introduction

Censorship circumvention is an arms race between covert communication systems and nation-state adversaries who use techniques such as IP and protocol blocking and deep-packet inspection [11, 26, 42] to prevent Internet users in their regions from accessing censored online destinations.

The ultimate goal of censorship circumvention is *unobservable communication*, i.e., the system should hide the fact that communication is taking place even if the adversary (e.g., an ISP with censorship filters installed) controls the entire Internet in the user's region. End-to-end encryption of network traffic is not sufficient because it hides neither the endpoints of communication, nor features such as packet counts, sizes, and timings. An unobservable communication system must be robust against adversaries who (a) whitelist permitted applications, and (b) deploy network traffic analysis to recognize and block connections that do not belong to any of the known, whitelisted applications.

A key challenge is how to design *a communication system whose network traffic cannot be reliably distinguished from the traffic of a known, popular application*. In general, there is a tradeoff between the performance and capacity of a covert channel and its resistance to traffic analysis. Circumvention techniques that operate low in the system stack and imitate existing network protocols have relatively high capacity but can be easily distinguished due to various discrepancies in their mimicry [15]. Steganographic techniques that operate high in the stack and encode covert communications into existing application content [7, 18] have low capacity. Our goal in this paper is to find a sweet spot in the system stack that provides (1) sufficient capacity for practical tasks such as Web browsing and audio streaming, as well as (2) resistance to traffic analysis.

Current state-of-the-art systems for unobservable communications, such as Protozoa [2] and Balboa [31], run a cover application—for example, video streaming—and substitute its traffic with covert content in the transport layer. Their prototype cover applications follow simple state machines, and the application content is either non-interactive [2], or deterministic and known to both communicating parties in advance [31].

Our contributions. We design, implement, and evaluate TELEPATH, a covert communication system based on Minecraft. Minecraft provides an interesting platform for exploring the design space of unobservable communication. (1) Minecraft is widely available and very popular in many countries that engage in pervasive Internet censorship, such as China. (2) Communications between Minecraft clients and servers are end-to-end encrypted. (3) It is common for users to run their own Minecraft servers, i.e., the game is decentralized. (4) There is an active community of users who modify Minecraft clients and servers, thus modifications to the Minecraft software are not unusual and, in fact, supported via popular "mod" platforms.

First, we identify application-specific causal dependencies between network packets as the reason why traffic substitution in the transport layer [2, 31] may produce traffic traces that are impossible in a real game. This motivates the need for *non-disruptive* content substitution that does not

DOI 10.1109/SP46215.2023.00183

introduce inconsistencies between the client and the server.

Second, we identify generic categories of non-disruptive game content, such as visual-effects data, map updates, and synchronization messages, and explain how TELEPATH uses this content in Minecraft for non-disruptive substitution.

Third, we demonstrate that TELEPATH resists traffic analysis. The best previously proposed classifiers perform similarly to random guessing when trying to distinguish traffic generated by TELEPATH from that generated by a Minecraft bot (a popular way to automate gameplay).

Fourth, we evaluate performance of TELEPATH on practical tasks such as browsing news websites and audio streaming. TELEPATH achieves a clientbound bandwidth of 1300 Kbps and serverbound bandwidth of 1-2 Kbps, which is sufficient for the browsing of many censored news websites and audio streaming at up to 256 Kbps bitrates.

2. Background

We focus on covert communications in the presence of a network-based adversary (e.g., an ISP controlled by a government censor). In contrast to systems [8, 10, 36, 38] that aim to hide who is communicating with whom but not the fact that communication is taking place, our goal is *unobservability*, i.e., to hide that the system is being used.

2.1. Censorship circumvention

The main challenge in constructing covert channels based on a new network protocol is that the resulting traffic is likely to be distinguishable from other network traffic. First, the adversary may run an implementation of the same channel, generate traffic traces, and build tools that recognize such traffic. Second, the adversary may whitelist known applications and block all traffic that does not look like it was generated by a whitelisted application.

Previous circumvention systems used several approaches to try and look like a known, whitelisted application (see Fig. 1): (a) imitate an existing protocol like Skype or HTTP [25, 41]; (b) send covert traffic through the network layer of an existing application; (c) run an existing application but substitute covert traffic in the transport layer [2, 31]; or (d) encode covert traffic into application-level content, such as audio [16] or video [19] streams.

In general, the closer to the network layer is the covert traffic injected, the easier it is to detect. Traffic that mimics other network protocols can be recognized via tell-tale discrepancies between implementations [15]. High-fidelity imitation of a complex application requires mimicking its correct behavior in any situation (including dynamic responses to network conditions), as well as every bug and quirk of its implementation. Injecting covert traffic higher in the software stack reduces the available bandwidth and can still be vulnerable to statistical traffic analysis [1].

2.2. Minecraft

Minecraft [21] is a 3D open-world sandbox game developed by Mojang Studios. The player interacts with a world composed of "blocks" that represent different items and materials. Users play any way they want, there is no required goal. Consequently, gameplay is diverse. Typical game activities include gathering resources such as food and ore, fighting monsters, and building houses.

Minecraft supports server-based multiplayer mode, which lets players work, build, or fight together in the same world hosted on a dedicated server. Multiplayer Minecraft is decentralized: players can deploy third-party servers using the server binary provided by Mojang. Network traffic between Minecraft servers and clients is encrypted.

Mojang's EULA permits modifications that introduce new game content or gameplay features. The Minecraft "mod" community has reverse-engineered the game's Java source code and developed over 100,000 mods [9]. For example, Hypixel, the world's most popular public Minecraft server, is heavily modded and supports many multiplayer minigames that are not available in the vanilla server. The game's communication protocol [23] is well-documented, enabling customization of client-server communication.

Minecraft can be automated using *bots*, which are mods that take control of the player and automatically perform tasks. Since players in the game's survival mode have to collect a large amount of resources to keep alive, bots are commonly used to automate repetitive tasks such as farming and mining. Although Minecraft traffic is hard to model in general due to very diverse player behaviors, it is possible to repeatedly run a specific bot to generate bot traffic.
In this paper, we will use the following Minecraft terms.

Block. Blocks are the basic units of structure in Minecraft. They are unit cubes arranged in a 3D grid, representing materials and resources such as stone, dirt, water, and ores. Blocks can be collected, placed, or destroyed by the user.

World/Map. The Minecraft world is the entire game area composed of blocks. It emulates the real-world terrain and biome. The core gameplay is to explore the world and interact with the blocks in it. The world is 30 million blocks long, 30 million blocks wide, and 256 blocks high.

Entity. Entities are the dynamic, moving objects throughout the Minecraft world, such as characters, animals, and other game items. An entity has the following properties: position (its coordinates in the grid), rotation (its facing direction), and velocity (its moving direction and speed).

Player. The player is a special entity controlled by the user. It can move around the world, interact with other entities, mine blocks, and place them elsewhere. If the player takes damage that exceeds its "health" property, it dies and cannot perform any action until it respawns.

Client. The game client is the software that controls the player. The client renders the game's graphical content (the player and the world near it) and displays it to the user. The client does not store any world or player data. It receives this data from the server and sends the player's updates (e.g., position, mining or block-placing actions) to the server.

Server. The game server is the software that lets multiple users play the game with each other online. The server hosts the Minecraft world, stores the position and status of each

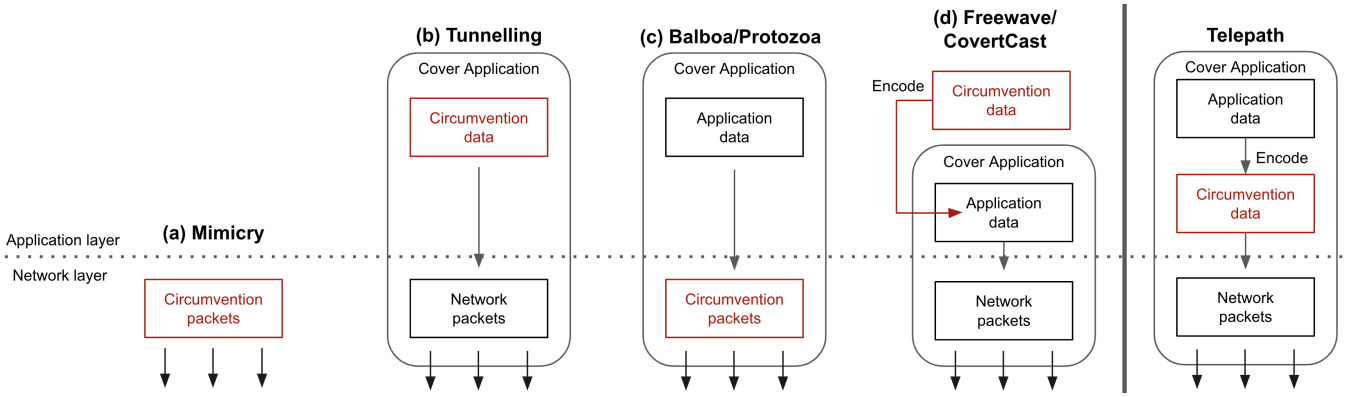


Figure 1: Circumvention approaches at different levels of the system stack.

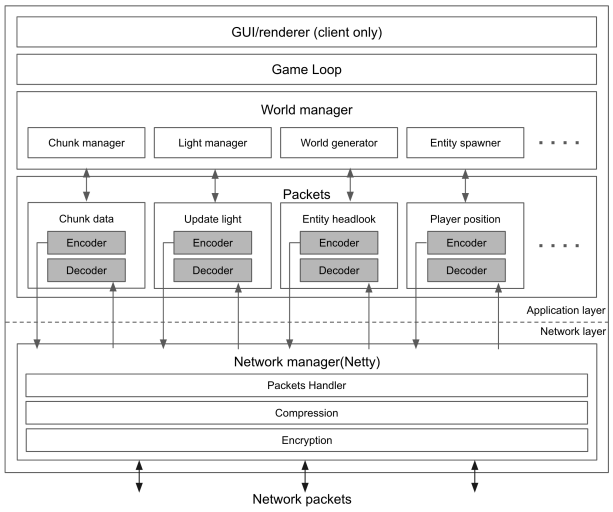


Figure 2: Minecraft software stack (shaded components are added by TELEPATH.)

block and entity, sends required data (e.g., map information and entity status) to the clients, receives all updates from the players, and applies them to the local game data.

Fig. 2 shows the software stack of Minecraft.

Chunk. Chunks are 16-block-long, 16-block-wide, 256-block-high[1] segments of a Minecraft world. They are the basic units of map operations (such as generation and loading), dividing the world into manageable pieces. Since the Minecraft world is too big to be loaded at once, the game only loads the chunks that are necessary to make it playable. Usually the server loads the chunks within a certain radius of the player and sends them to the player's client. Because only these chunks can be displayed by the client, this is the player's "view distance" (determined by the server).

3. Threat model

We assume a state-level censor who can monitor, store, inspect, and block the network traffic of all users within its scope. It can block connections to prohibited IP addresses and use deep packet inspection to filter unencrypted traffic containing blacklisted content. We assume the censor can identify network protocols and block prohibited protocols such as Tor. Furthermore, we assume the censor *whitelists* permitted applications and protocols, and blocks any traffic that does not look like it was generated by a permitted application and conveyed over a permitted protocol.

We assume the adversary cannot subvert TLS and thus cannot decipher encrypted network packets. When observing encrypted traffic flows, the adversary can observe packet sizes and timings, as well as any unencrypted metadata. The adversary may build statistical traffic models [1] to distinguish covert traffic from genuine application traffic.

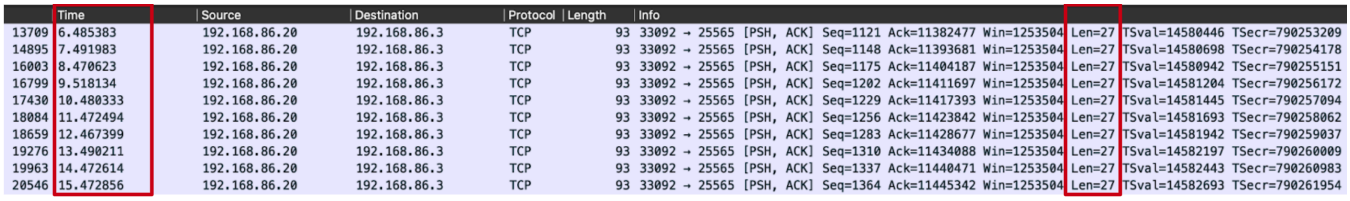
We assume the adversary knows how whitelisted applications operate and can thus infer partial application states from the encrypted traffic and detect if the application enters an impossible state. With games, the adversary can (a) infer some game events from packet sizes and timing, and (b) check simple invariants that these events must satisfy. In Section 4, we use Minecraft to show how game events and invariants can be inferred from encrypted network traffic.

4. Minecraft traffic invariants

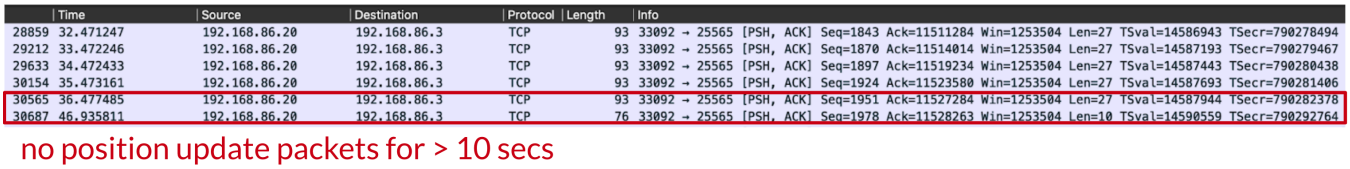
In this section, we demonstrate that certain Minecraft game events can be inferred from the encrypted network traffic by recognizing packets with unique lengths. We then identify simple invariants satisfied by any Minecraft game session. The corresponding invariants over encrypted packet traces can be checked by a network observer via deterministic, single-trace tests without any statistical traffic analysis.

We then show that circumvention approaches that substitute content using deterministic templates (such as [31]) do not work for Minecraft. Even if the templates are obtained from actual Minecraft games, content substitution can produce traffic traces that violate the above invariants and are consequently easy to recognize, since they could not have been generated by a real Minecraft session.

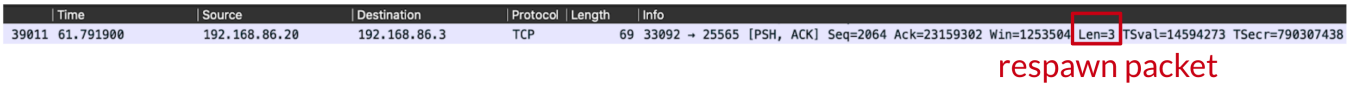
1. Minecraft Version 1.15.2



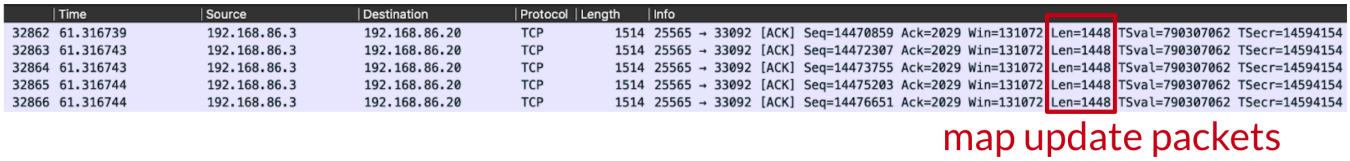
(a) Player position update packets



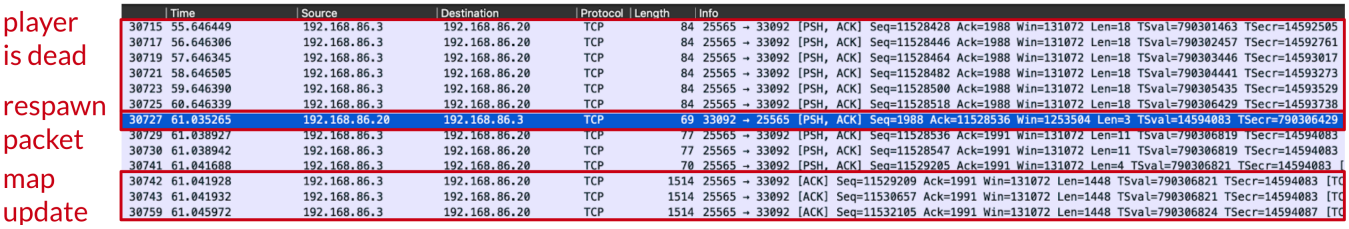
(b) Player is dead



(c) Respawn packet



(d) Map update packets



(e) Valid game trace

Figure 3: Network traces corresponding to different Minecraft in-game events.

4.1. Game events

This section is not intended to be an exhaustive list of events that can be inferred from encrypted Minecraft traffic. Its purpose is to demonstrate the existence of *some* packet invariants that (a) must be satisfied by any Minecraft packet trace (Section 4.2), and (b) are violated by deterministic content substitution (Section 4.3).

Player dead. While the player is alive in the game, the game client periodically sends information about its position to the game server. If the player is idle, the frequency of these updates is about 1 packet per second, more if the player is moving. When the player is dead, the game client does not send any position updates until the player respawns. Therefore, if no position update packet is observed for a while, a network observer can infer that the player is dead.

Position update packets can be identified via their sizes. There are two types of position update packets in Minecraft: PositionPacket and its subclass PositionRotationPacket. Both contain `double` data fields X, Y, Z representing the player's coordinates, and a `boolean` field OnGround, indicating if the player is on the ground. PositionRotationPacket also contains `float` fields Yaw and Pitch, representing the player's direction. Two bytes of type information are appended during packet encoding. Therefore, a PositionPacket is $3 \times 8\,(X, Y, Z) + 1\,(OnGround) + 2\ (type) = 27$ bytes long, a PositionRotationPacket is $27 + 2 \times 4\,(Yaw\ and\ Pitch) = 35$ bytes long. These packet lengths are unique. While the player is dead, the game client does not send any 27- or 35-byte packets. Figure 3a shows position update packets in a Minecraft network trace.

A network observer can infer that a player is dead if it does not observe client-to-server 27- or 35-byte packets for

a while. Figure 3b shows the network trace of a dead player.

Player respawned. Upon the player's death, a "You died!" screen appears and the user cannot do anything until they click the "Respawn" button. The click sends a 3-byte respawn request packet to the server; upon receipt, the server sets the player alive. This packet length is unique: the client does not send any other 3-byte packets while the player is dead. If a network observer observes a 3-byte packet, it can infer that a dead player has respawned (see Figure 3c).

Map update. The game server sends map updates to the client when the player is alive and moving (including respawning, since the player is teleported to the spawn point). Map-update packets are the only packets in the normal gameplay longer than 1000 bytes (see Figure 3d).

4.2. Network trace invariants

Any Minecraft session satisfies these invariants:

Invariant 1. ***Once dead, a player stays dead until it respawns.***

Invariant 2. ***A dead player does not receive map updates.***

It is easy to check if an encrypted Minecraft traffic trace satisfies these invariants. Figure 3e shows a network trace generated by unmodified Minecraft. Observe that the player is dead at the beginning of the trace since there is no player position packet. Then the client sends the respawn packet to the server. After receiving the respawn packet, the server begins to send information to the player, which includes map update packets. The invariants are satisfied.

4.3. Breaking deterministic content substitution

To show that deterministic content substitution can produce traces that are impossible in a real game, we implemented a strawman design of TELEPATH. It is similar to Balboa [31], which uses a deterministic model to shape network traffic to look like unmodified application traffic.

Our strawman design uses static traffic models (*templates*) generated from actual Minecraft sessions. We run multiple, unmodified "shadow games" in advance. On both the client and server side, for every function call from the game logic to the network manager, we store the timestamp and size of the arguments, thus forming a template. To send covert data, the circumvention proxy follows one of the templates but replaces the message content with covert data.

On both the client and server side, for every pair of function calls C_1 and C_2 that the shadow game made to its network manager, the circumvention client or server makes C_1' and C_2' such that (1) the called functions are the same, (2) the time between C_1' and C_2' is the same as the time between C_1 and C_2, and (3) the size of the arguments is the same. This design ensures that every message sent through the network manager could have been generated by some real game activity in the shadow game (except for the actual encrypted content). As in [31], the goal is indistinguishability under passive traffic analysis.

Unfortunately, this approach fails. Although our strawman design generates traffic that is statistically indistinguishable from some shadow game, individual traffic traces sometimes violate the invariants from Section 4.2. The key problem is that, like in [31], traffic templates are application-agnostic. They produce messages of the right size at the right time but have no information about the game state. Therefore, client-server communication in the circumvention system does not respect **causal relationships** between messages. Even a slight, natural delay in the network can violate these relationships, resulting in an observable violation of trace invariants based on the game semantics.

Figure 4 shows a network trace generated by the strawman design that violates the two invariants from Section 4.2. Observe that the player is dead at the beginning of the trace since there has been no position update packet for a relatively long time. However, the server sends a map update packet (#31213) *before* it receives the respawn packet (#31215), which is impossible in any real Minecraft session.

Therefore, template-based content substitution in Minecraft is easy to recognize due to occasional violations of trace invariants when packets are out of order, which is common in real network environments. These violations can be detected with simple, passive, single-trace tests. An active adversary may artificially introduce longer network delays and/or re-order packets, which will cause more observable violations of trace invariants.

5. Design of TELEPATH

TELEPATH is a content substitution system. Content substitution involves running a cover application binary with normal inputs and replacing the payloads of its network messages with covert data at the application or transport layer. Previously proposed cover applications have simple finite-state machines (e.g., audio/video streaming or HTTP) and deterministic inputs (audio/video files, Web pages).

Video games such as Minecraft, however, have complex action space and a huge number of possible application states. They also take nondeterministic inputs such as mouse movements and keystrokes. As we show in Section 4.3, existing content substitution approaches operate at the transport layer and cannot be applied to game-based channels. They may produce traffic traces that violate known game-trace invariants and are thus easily detectable.

The key idea behind TELEPATH is *non-disruptive content substitution*. Instead of substituting all application content, TELEPATH substitutes only messages that do not cause long-lived inconsistencies between the client and the server.

5.1. Identifying non-disruptive content

We identify generic categories of non-disruptive content, applicable to many games, and explain how to substitute them. Formal definitions for each category can be found in Appendix A, modeling client and server as state machines. Server sends current state to clients, clients perform actions resulting in updates, server collects update messages, applies them to the state, sends state to clients, etc.

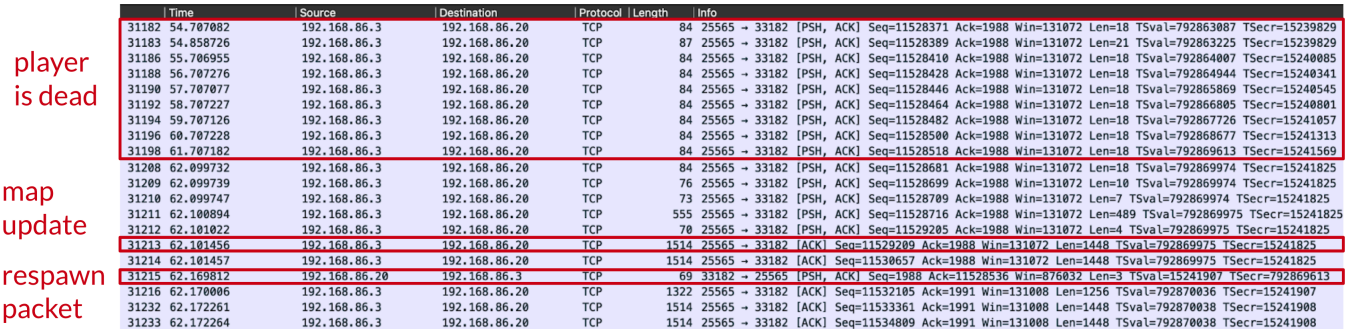


Figure 4: An impossible trace generated by the strawman design of TELEPATH. The server sends a map update packet to a dead player before it receives the respawn request, which violates trace invariants based on the game semantics.

Visual effects. A feature of the game state is a visual effect if client actions generate the same update message regardless of its value. Examples are unreachable map areas, animation information, and projectiles that don't hit the player.

Identifying visual effects requires game-specific analysis. A generic heuristic is to agree during setup that the client will *not* perform certain actions. The content with which the client will not interact can be substituted. For example, in RPG games like Diablo 2, a player may pick up only rare looted items and ignore others (a common scenario when farming for valuable items). Items sent by the server but never used by the client can be treated as visual effects.

Synchronization messages. Most online games rely on periodic updates to synchronize clients and servers. An update corresponding to a periodic action performed by the client or the server is a synchronization message.

Variable update frequency is a standard technique in client-server online games to improve performance [12, 37]. Reducing the frequency of updates doesn't cause long-lived inconsistencies between the client and the server. Therefore, the client and the server can exchange update messages with the original frequency but substitute some of them with covert content (thus reducing the effective update frequency, as far as the game is concerned). For example, player position updates (inherent in video games) are a type of synchronization message that can always be used for non-disruptive substitution by reducing update frequency.

The content of omitted updates can often be predicted based on the previous updates. Player prediction is a standard technique to compensate for network latency in fast-paced online games [5, 6]. Upon receiving a periodic update whose content has been substituted, the system can estimate the value of the substituted content, and the game can use these predictions to keep its state up-to-date.

5.2. Architecture of TELEPATH

Figure 5 shows the main components of TELEPATH. The **local proxy** on the client side is a standard SOCKS5 proxy that accepts requests from applications such as Web browsers and forwards them to the TELEPATH mod inside the Minecraft client. The **remote proxy** on the server side handles requests from the client. It opens connections to the requested addresses and forwards the responses to the TELEPATH server mod. The **client (server) mod** is a modification to the Minecraft client (respectively, server). It consists of an encoder and a decoder and implements a bidirectional point-to-point covert channel.

Encoder. The encoder encodes requests from the local proxy (respectively, responses from the remote proxy) into game messages, which are sent through the game's network manager. Figure 6 shows the architecture of the encoder. The key component is the interceptor that filters all messages between the game loop and the network manager, looking for non-disruptive messages. Once it intercepts a non-disruptive message, it discards its content and substitutes covert data from the local proxy. The rewritten message is then passed to the network manager like any other game message.

Decoder. The decoder (see Figure 7) identifies incoming packets that contain covert data. Upon receiving a packet, the network manager dispatches it to the corresponding handler, where the interceptor determines if it contains covert data, extracts the data, and passes it to the proxy. The interceptor also notifies the dummy message generator to generate suitable dummy content for the messages whose content was replaced by the covert data.

Next, we describe the messages substituted by TELEPATH and explain (a) why they are non-disruptive, and (b) how to safely substitute their content and generate appropriate dummy content without causing long-lived inconsistencies between the client and server states.

5.3. Clientbound non-disruptive messages

Light update. Light is the attribute of Minecraft blocks that affects visibility, mob spawning, and plant growth. A block's light level is an integer between 0 (darkest) to 15 (brightest). Only blocks with non-zero light level are visible to the player. Mobs only spawn when light level is low and plants only grow when light level is high. The light level of a block is calculated by the server and sent to the client via a light update message containing the light information of all blocks in a single chunk. Light update messages account for about 33% of Minecraft's clientbound bandwidth.[2]

2. Numbers are based on our farming bot described in Section 7.1.

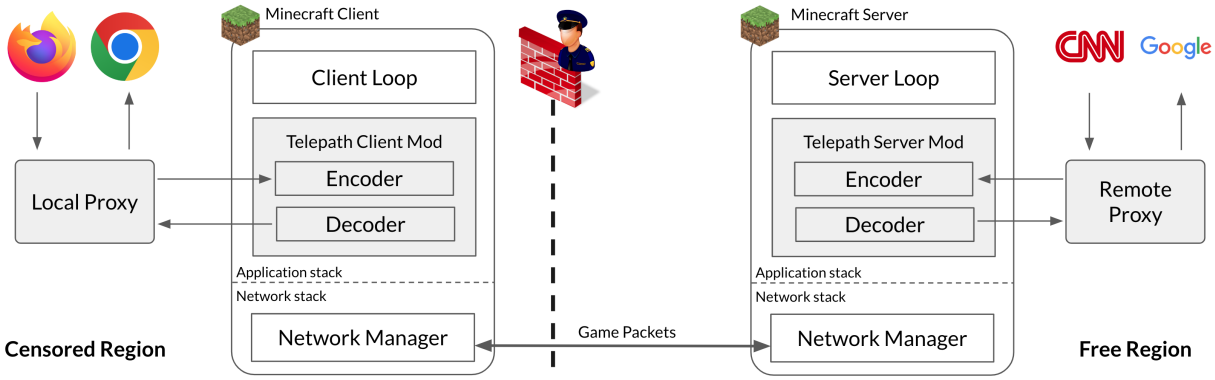


Figure 5: Architecture of TELEPATH. The components of TELEPATH are highlighted in a darker shade.

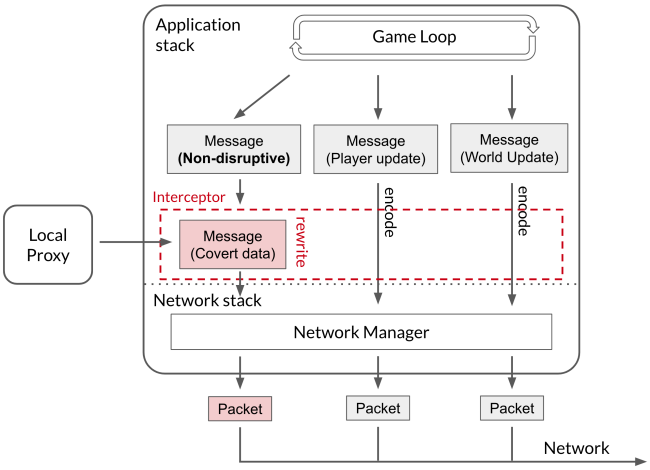


Figure 6: TELEPATH Encoder.

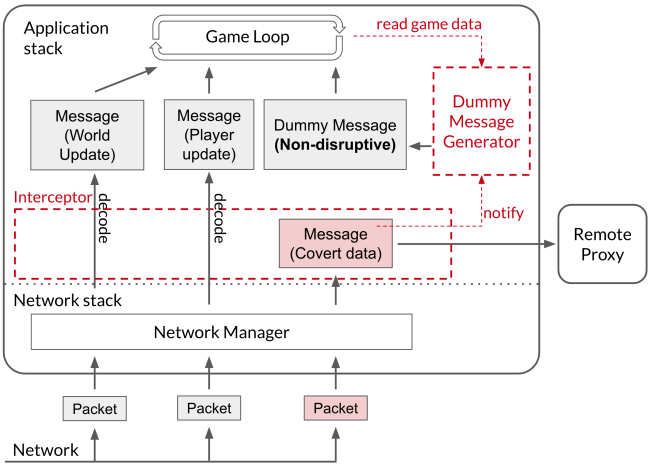


Figure 7: TELEPATH Decoder.

Since mob spawning and plant growth are processed by the server, light level only affects visual effects such as rendered brightness and visibility of the blocks on the client side. Therefore, it is safe to set the light level of all blocks to 15 (brightest) on the client side without causing inconsistencies between the client and server states.

Figure 8: Light update message layout. Red denotes data fields replaced by covert data.

Figure 9: Chunk data message layout. Red denotes data fields replaced by covert data.

Figure 8 shows the data layout of a light update message. X and Z represent the location of the chunk, the masks indicate if a block in the chunk has data in the following light array. The list of light arrays contains the actual light level information. Each 2048-byte array represents the light level of a 16x16x16 area, in which each byte represents the light levels of 2 blocks (e.g., levels of 5 and 15 are stored as 0x5F). To substitute the content of a light update message, the encoder simply discards the data in the light arrays and writes in the covert data. Upon receiving such a message, the decoder extracts the covert data from the light arrays and the dummy content generator fills the arrays with 0xFF, setting the light level of all blocks to 15.

Chunk data. When the player moves to a new location, the server loads the chunks (see Section 2.2) within the player's view distance into its memory and sends them to the client to display to the user. Chunks outside the player's view distance are unloaded. A chunk data message contains information about all blocks in a chunk. These messages account for about 55% of the clientbound bandwidth.

Unlike the light level, which only affects visual properties at the client side, chunk data contains information about the blocks that the player can interact with. Naively substituting all chunk data would cause inconsistencies between the client and the server. For example, if blocks are randomly assigned to a chunk, the player may stand on an air block, which is invalid. In Minecraft, however, the player can only interact with blocks or entities within a certain range that is usually smaller than its view distance. Some

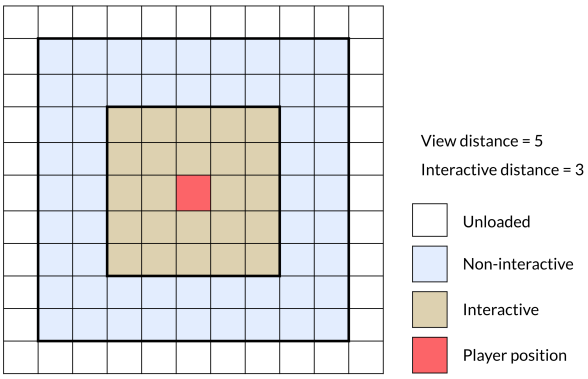


Figure 10: Player's interactive distance.

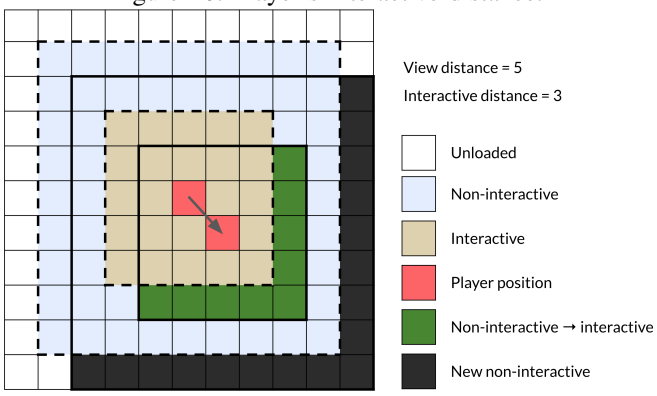


Figure 11: Changes in interactive distance.

common ranges are (1) the *attack reach*: the player can only attack entities within 3 blocks, (2) the *building reach*: the player can only place or collect blocks within 4 blocks, and (3) the *mob detection range*: mobs within this range, which depends on the type of mob and has the maximum of 64 blocks (Enderman), can detect and try to attack the player.

We define the *interactive distance* of a player as the range that the player can interact with. We divide the loaded chunks within the player's view distance into two categories: interactive chunks, i.e., the chunks within the interactive distance, and non-interactive chunks, i.e., the chunks that the player can view but cannot interact with. Since non-interactive chunks have visual effects only, the corresponding chunk data messages are non-disruptive and can be used to transmit covert data. In our prototype of TELEPATH, we set the interactive distance to 5 chunks (80 blocks), 1 chunk more than the maximum mob detection range. Figure 10 shows the interactive distance of a player.

Since both interactive chunks and non-interactive chunks are sent in chunk data messages, the receiver cannot distinguish between them without additional information. To mark non-interactive chunks that carry covert data, we utilize free bits in the chunk's X coordinate. This coordinate is a 32-bit integer with a hard-coded [-1875016, 1875016] limit due to the world boundary in the game. Therefore, we can encode both the chunk's X coordinate and the covert bit into the same integer. Figure 12 shows this encoding.

Figure 9 shows the data layout of a chunk data message. The encoder first sets the covert bit in the X coordinate of a non-interactive chunk, then overwrites all fields except the X and Z coordinates with covert data. Upon receiving a chunk data message, the receiver first decodes the X coordinate to check if the covert bit is set. If not, the chunk is processed by the original game code and displayed to the user. If the covert bit is set, the decoder extracts covert data from the message and requests a dummy chunk from the dummy content generator. The dummy chunk is randomly selected from the loaded interactive chunks on the client side.

Handling changes in interactive distance. Whether a chunk is interactive or not depends on its distance from the player, which changes as the player moves around (see Figure 11). A non-interactive chunk may become interactive. If a chunk was used to send covert data, it is considered loaded at the client side, thus the server won't send it again.

To avoid inconsistencies between the client state and the server state, TELEPATH needs to send to the client the chunks whose status has changed from non-interactive to interactive without introducing extra messages. This means they cannot be sent via chunk data messages. We solve this conundrum by treating these newly interactive chunks as covert data. When the player moves to a new position, the server checks if any chunk has changed from non-interactive to interactive. If such a chunk is found, the server encodes the chunk data into a byte array and adds it to the update queue. When a non-interactive chunk data message is intercepted, the encoder first checks the update queue. If the queue is not empty, the encoder reads the encoded chunk byte array from the queue and sends it as covert data. Otherwise, the encoder sends covert data from the local proxy. Similar to the covert bit above, we use another free bit in the chunk's X coordinate (see Figure 12) to indicate that the covert data is a chunk that changed its interactive state, as opposed to the data requested by the local proxy.

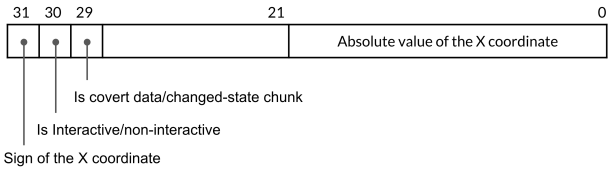


Figure 12: X coordinate encoding.

Entity head look. Entity head look is a clientbound message with the head direction of a living entity such as player, animal, or mob. Head direction is a visual effect; the only exception is an Enderman, a monster that can be provoked by a player looking them in the eyes (i.e., the player's head direction is towards the enderman). Generation of monsters including endermen can be disabled by setting the game difficulty to peaceful, thus the entity head look message is non-disruptive in peaceful games.

Figure 13 shows the data layout of the entity head look message. The encoder discards the head direction field and substitutes covert data. Upon receiving such a message, the decoder extracts the covert data and the dummy content generator randomly assigns a head direction to the entity.

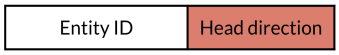


Figure 13: Entity head look message layout. Red denotes data fields replaced by covert data.

5.4. Serverbound non-disruptive messages

Hand animation. Hand animation message is a simple message used to notify the server that the player is swinging its hand. The message contains a single byte indicating which hand. As the message name suggests, it carries visual-only information and is consequently non-disruptive. The interactions between the player's hand and other blocks or entities are sent through other types of messages.

To encode covert data into a hand animation message, the encoder simply writes 1 byte of covert data into the hand-indication byte. After decoding the message, the dummy content generator sets the byte to right hand.

The hand animation message is the sole visual-only serverbound message in the game. It accounts for only about 2% of the serverbound bandwidth. To increase the capacity of the serverbound channel, it is necessary to leverage messages that carry non-visual information.

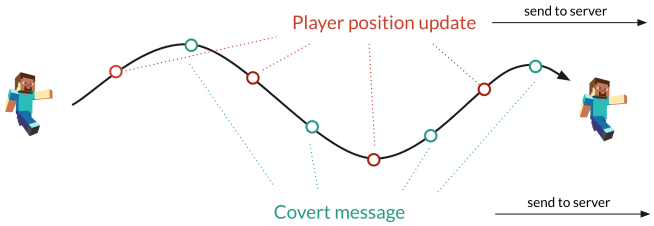


Figure 14: Player position interpolation.

Player position. Every game tick, the client checks the player's position and sends a player position message to the server if the player has moved. It contains the player's X, Y, and Z coordinates and the facing direction. These messages account for about 80% of the serverbound bandwidth.

Since the player position message changes the server state, naively substituting these messages will cause inconsistencies between the client and the server. For example, suppose the player moves to a new location and starts digging a block. If all player position messages are substituted, the server doesn't know that the player has moved. When the server receives a message that the player is digging a block far from its last known position, an inconsistency ensues.

Because player position messages are periodic updates, we can lower their frequency and estimate positions that have been replaced by covert data. We leverage the fact that player movement is continuous. The player cannot move too far within one game tick to result in an observable inconsistency between the server and the client. The player's position can also be estimated from their recent updates. We interleave unmodified position messages and substituted messages so that the server can keep the player's position up-to-date without long-lived inconsistencies with the client. The server uses interpolation (see Figure 14) to estimate the player's position between two unmodified updates to make up the information lost to substitutions.

We consider two interpolation methods to estimate the player's position based on its recent movements. Let $\vec{p_1} = (x_1, y_1, z_1)$ be the most recent position update from the player, $\vec{p_2}$ the penultimate update, and $\vec{p'}$ the position we want to estimate. We assume that the intervals between game ticks are consistent. Since we interleave unmodified position updates and covert messages, the time between $\vec{p_1}$ and $\vec{p_2}$ and between $\vec{p'}$ and $\vec{p_1}$ is 2 ticks and 1 tick, respectively.

- **(Nearest neighbor)** $\vec{p'} = \vec{p_1}$
- **(Linear)** $\vec{p'} = \vec{p_1} + (\vec{p_1} - \vec{p_2})/2$

Both methods work well in our prototype and don't produce observable inconsistencies.

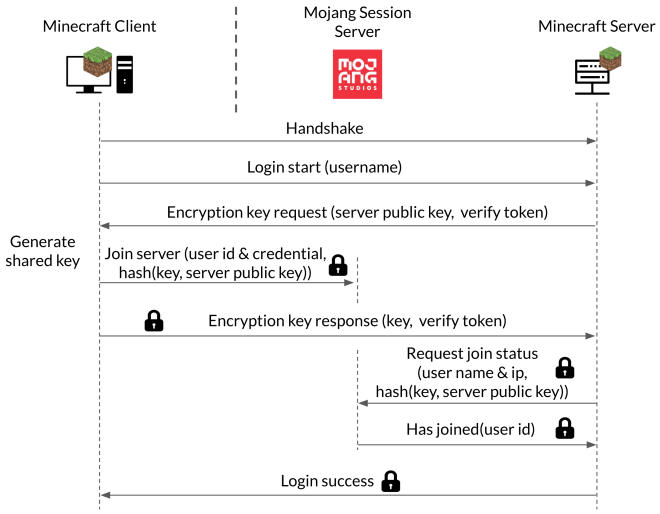


Figure 15: Minecraft connection establishment.

5.5. Bootstrapping

Figure 15 shows how connections are established between Minecraft servers and clients.

Key exchange. TELEPATH follows the unmodified Minecraft key exchange protocol to establish an encrypted channel. The client first sends a handshake message and a login start message with the player's username to the server. The server replies with an encryption key request message that contains the server's public key and a random verify token. The client generates a shared symmetric key and sends an encryption key response message with the shared key and the server's verify token, encrypted under the server's public key. Upon receiving this message, the server notifies the client that the channel is established.

Server authentication. In Minecraft, the server's public key is randomly generated when the server starts. Since a client cannot authenticate the server, a censor could mimic a TELEPATH server to identify TELEPATH users. We add server authentication to TELEPATH. Instead of randomly

generating the server's public key, TELEPATH selects it from a set of public keys known to both the server and the client. During key exchange, the client verifies that the server's public key is in this set. Otherwise, the client does not start a TELEPATH session and behaves like an unmodified client.

Client authentication. We leverage Minecraft's client authentication mechanism to verify that the client is a TELEPATH user. Each Minecraft user has a unique UUID (Universally Unique Identifier) assigned by Mojang when the game is purchased. To connect to a Minecraft server, the client first logs into the user's Mojang or Microsoft account (depending on the account type) and obtains a credential called the access token. During the key exchange, the client sends a join-server message, which includes the user's UUID, the access token, and the hash of the (shared key, server public key) tuple to the Mojang session server. The Mojang server verifies the access token, keeps a record that the user is trying to connect to a server, and stores the hash. After the Minecraft server receives the encryption key response message from the client, it requests from the Mojang session server the join status of the user with the username and the hash of the key tuple. If the username and the hash match the record in the Mojang session server, the latter replies with a has-joined message and the user's UUID, so the Minecraft server can verify the identity of the connecting client. All connections to Mojang or Microsoft are based on TLS, thus it is not possible for a censor to either mimic Microsoft or Mojang, or obtain the user's credentials.

To verify that a client is a TELEPATH user, the server stores a list of UUIDs of valid users. When a client successfully connects to the server and finishes the user authentication process, the server checks whether the user's UUID is on the list. If so, the server starts a covert session. Otherwise, the server behaves like an unmodified server.

6. Implementation

Minecraft is closed-source software, but its end-user license agreement (EULA) allows users to modify both the client and the server under certain restrictions: (1) the mod cannot contain a substantial part of Minecraft's copyrightable code or content, and (2) mod developers can distribute mods but not modded Minecraft implementations.

Most existing Minecraft mods are based on mod loaders. A mod developer can use the loader's API to create and distribute mods without changing the Minecraft code directly. Mod loaders, on the other hand, modify Minecraft. To comply with the EULA, loaders are distributed as code patches, which do not contain a substantial part of Minecraft's code.

TELEPATH changes several aspects of the Minecraft implementation, such as the map loading mechanism. Because these changes cannot be done via the API of any existing mod loader, we implemented TELEPATH by modifying MinecraftForge [22], a popular, open-source loader. This enabled us to package the changes into a patch to the Minecraft code, which does not violate the EULA.

Our proof-of-concept implementation of TELEPATH involves approximately 2500 lines of Java code, including a modification to MinecraftForge 1.15.2-31.2.50 and a SOCKS proxy implementation. To ensure that compression rates are not different between the real game messages and covert messages, we turn off the compression of network traffic by setting the compression-threshold to -1 in the server configuration (this option is somewhat unusual but is used in actual game servers to optimize their performance). We also adopt an open-source Minecraft bot (see Section 7.1) to automatically perform in-game activities that can be used by TELEPATH to transmit covert traffic.

Implementation of Web browsing and audio streaming benchmarks is described in section 7.4.

7. Evaluation

We evaluate TELEPATH by measuring (a) how it resists to statistical traffic analysis, and (b) how it performs for Web browsing and audio streaming.

7.1. Experimental setup

Unless otherwise indicated, the Minecraft client is running on an Intel i7-5960X CPU @ 3.00GHz with 16 cores and 96 GB of memory, the Minecraft server on an AWS EC2 t3.xlarge instance with an Intel Xeon Platinum 8259CL CPU @ 2.50GHz with 4 cores and 16GB of memory. Both the client and the server are running in docker containers. The client is running in an Xvfb [44] virtual display with 1024x768 resolution. We use a Python script with PyAutoGUI [28] to automate the interaction with the game's GUI. The server is running in the headless mode.

We are not aware of any representative datasets or plausible models of real-world Minecraft traffic. Therefore, we designed TELEPATH to be indistinguishable from a specific Minecraft bot. TELEPATH does not aim to resist detection of automated behavior. Automated gameplay is common in Minecraft because resource collection is time-consuming and repetitive. For example, to obtain a diamond gear set (best in the game) in survival mode, the player must mine over 30,000 blocks, requiring 5 hours of repetitive mining. Many Minecraft players thus use bots to collect resources. If censors block automated gameplay, it would cause collateral damage and drive many users from the game.

We use Minebot [20], a Minecraft mod that controls the player and automates its activities, for a simple farming bot which repeatedly searches for the nearest fully-grown wheat crop, harvests it, and places a wheat seed on the same tile. We constructed a survival farm (see Fig. 16) for this bot to operate on. It includes a 44×44-block wheat farm, 22×44 sugar-cane farm, 44×23 cow farm, and 22×23 sheep farm. Each individual farm is surrounded by wood fences to restrict the tiles that the bot can interact with.

TELEPATH is based on the MinecraftForge mod loader [22], which sends extra messages between the client and the server to support its modding API. Therefore, to run the bot, we use Minecraft with unmodified MinecraftForge installed; otherwise, a traffic classifier may mistakenly use the extra messages to "distinguish" TELEPATH and the bot.

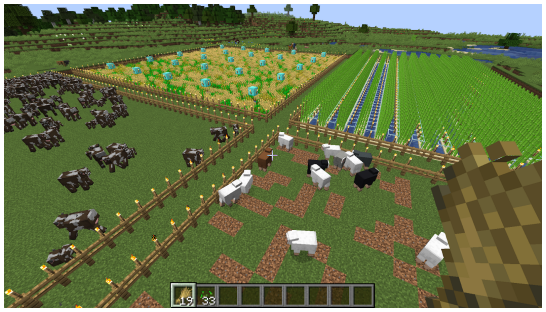


Figure 16: The Minecraft farm used in our evaluation.

7.2. Traffic analysis methodology

We adopt the methodology of [1, 2] to determine if a passive network observer can distinguish TELEPATH traffic from that of the farming bot from Section 7.1. Our results are thus a *conservative over-estimate* of how distinguishable TELEPATH would be in the presence of Minecraft traffic generated by diverse gameplays.

Following [1, 2], we extract two sets of features from the network traffic between the client and the server: (1) summary statistics of packet sizes and packet inter-arrival times, and (2) quantized packet sizes. We use these features to train random forest [4] classifiers because similar decision tree-based classifiers have been successful at identifying the traffic of previously proposed censorship circumvention systems [1]. For each experiment, we train the classifier using 10-fold cross-validation in Scikit-learn [27].

To measure if these classifiers can distinguish TELEPATH traffic from bot-generated traffic, we follow [1, 2] and calculate the true positive rate (TPR), false positive rate (FPR), and the area under the Receiver Operating Characteristic (ROC) curve. TPR is the fraction of TELEPATH traces that are correctly identified by the classifier among all TELEPATH traces. FPR is the fraction of bot traces that are erroneously classified as TELEPATH traces. ROC plots TPR against FPR at different thresholds. The area under the ROC curve (AUC) measures the performance of the classifier. A perfect classifier would achieve AUC=1.

We evaluate TELEPATH on datasets that are equally balanced between TELEPATH and bot traces. In a real-world deployment, covert communication would account for a negligible fraction of the overall Minecraft traffic, thus any non-zero false positive rate would overwhelm the adversary. In the balanced setting, however, even random guessing has AUC=0.5. Our reported AUC values are thus *conservative over-estimates* of the classifier's real-world performance.

We generate 100 TELEPATH traces and 100 bot traces in random order. For each run, our client connects to the game server, waits 40 seconds for the game content to load, then starts the bot. The bot runs for 60 seconds, then disconnects. We restart the client and the server after every run and reset the game world to prevent the large number of new items created by the bot from crashing the server.

For each TELEPATH run, we randomly select 1 website among Google, Wikipedia, CNN, BBC, NYTimes, and Reddit (all are commonly blocked by state-level censors), and execute Firefox to browse it through the TELEPATH proxy. We set the maximum loading time for each webpage to 60 seconds (pages load partially before they time out). Network traffic is captured using Wireshark's tshark [43].

7.3. Traffic analysis evaluation

Baseline. For the baseline experiments, we run the server in the AWS region closest to the client. Figure 17a shows ROC curves for our classifier when distinguishing TELEPATH traffic from the bot traffic. AUC=0.59 whether using summary statistics or quantized packet sizes, thus an adversary who tries to use this classifier to detect TELEPATH traffic would incur significant collateral damage. For example, when TPR is 0.6, FPR exceeds 0.5, i.e., to block 60% of TELEPATH traffic, more than half of the "genuine" bot traffic will also be erroneously blocked. If TELEPATH were a small fraction of the traffic, the false positive rate would be prohibitive.

The classifier slightly outperforms random guessing. We analyzed the per-feature distribution and found that the leakage mainly comes from the features related to packet inter-arrival times. We conjecture it is caused by delays due to the encoding of covert messages into game packets. We also conjecture that delays may slightly change the packetization of game messages. This minor leakage is not unusual [31] and, with a high false positive rate, does not provide a way to reliably identify TELEPATH traffic.

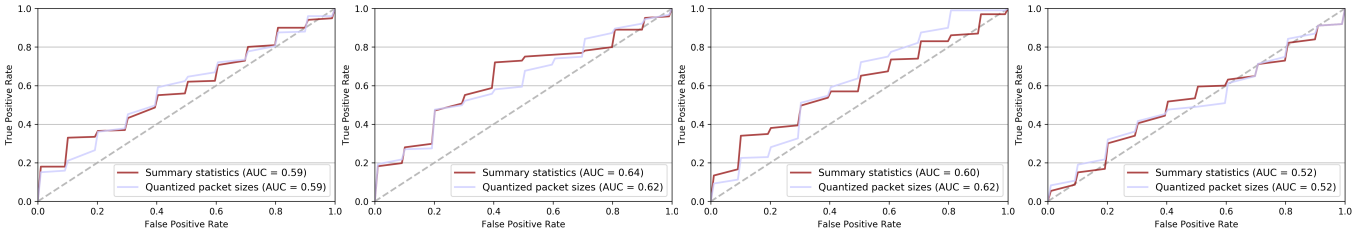
Next, we investigate how TELEPATH resists traffic analysis under different network conditions.

High latency. We use remote AWS servers in the US-west and Singapore regions to measure how TELEPATH resists traffic analysis under intra-continental and inter-continental network latency, respectively.

Figures 17b and 17c show ROC curves of our classifier. With summary statistics as features, the classifier achieves AUC of 0.64 (US-west) and 0.60 (Singapore) vs. 0.62 and 0.62, respectively, with quantized packet sizes. This is better than the baseline but significantly worse than a similar classifier achieves against Balboa [31] when network latency is not randomized. When latency is high, both bots and TELEPATH generate more packets ($\sim 2\times$) than the baseline. We conjecture that this magnifies the leakage caused by the encoding of covert messages.

Packet loss. An active adversary may artificially drop packets to cause a circumvention system to generate distinguishable network traffic. We evaluated TELEPATH under 2%, 5%, and 10% packet-loss rates, using *tc* [35] to introduce packet losses. Both TELEPATH and unmodified Minecraft client disconnect when packet losses reach 5%, thus an adversary cannot introduce heavy packet losses to distinguish TELEPATH from unmodified Minecraft.

Figure 17d shows the classifier's ROC curve under 2% packet loss. Both summary statistics and quantized packet sizes achieve AUC=0.52, below the baseline. TELEPATH and bots generate fewer packets ($\sim 1/3$) than the baseline, reducing the leakage.



(a) Baseline: local server (b) Remote server: US West (c) Remote server: Singapore (d) 2% packet Loss

Figure 17: Traffic analysis results

Website	US-east	US-west	Singapore	Japan
en.m.wikipedia.org	34.7 ± 8.5	35.8 ± 11.1	33.0 ± 9.7	35.6 ± 6.9
lite.cnn.com	43.9 ± 11.7	44.0 ± 9.8	39.8 ± 6.8	40.2 ± 11.2
bbc.com	Timeout	Timeout	Timeout	Timeout
nytimes.com/timeswire	67.1 ± 9.4	67.1 ± 8.0	60.0 ± 14.5	63.1 ± 11.3
cbc.ca/lite	43.4 ± 8.5	46.4 ± 9.3	53.0 ± 14.8	48.0 ± 7.3
dw.com/en	64.3 ± 7.5	76.0 ± 14.8	75.9 ± 12.2	69.6 ± 20.1
i.reddit.com	35.0 ± 6.7	41.9 ± 14.6	40.9 ± 7.3	37.2 ± 4.4
mbasic.facebook.com	7.1 ± 5.1	6.9 ± 3.5	7.0 ± 3.1	4.9 ± 3.3
bing.com/search?q=minecraft	18.1 ± 3.4	18.4 ± 4.7	18.7 ± 3.7	20.6 ± 3.4

TABLE 1: Webpage load time through TELEPATH proxy. Values are in "mean $\pm$ standard deviation (seconds)" format.

7.4. Performance evaluation

Covert messages in TELEPATH are sent using game messages that are closely related to the player's movement in the game (map updates, player position updates). Therefore, performance of TELEPATH highly depends on the player's behavior and has high variability. When using the farming bot described in section 7.1, TELEPATH achieves the clientbound throughput of approximately 1300 Kbps and the serverbound throughput of 1-2Kbps. We evaluate the usability of TELEPATH on practical tasks such as Web browsing and audio streaming.

Web browsing. We use a Firefox browser connected to the TELEPATH proxy to browse nine popular websites that are blocked by state-level censors in multiple countries. To emulate a real-world scenario where TELEPATH users are located in different countries, we ran clients on AWS machines in different regions (US-east, US-west, Japan, Singapore) and the server on an AWS machine in US-east. To emulate a multi-player gameplay, all clients connect to the server at the same time, and the bot is active in the background during browsing. We browse each page 5 times and measure the corresponding load times.

Since TELEPATH has low serverbound bandwidth, we use the low-bandwidth (mobile or lite) version of each website if available. To reduce the number of requests to the server, we block images and JavaScript (a real-world user can use a browser plugin to manually load some images and scripts). We cut off the loading at 120 seconds.

Table 1 shows the load time of each page in different regions. In all regions, 8 out of 9 pages loaded successfully, 6 of them in under 1 minute. `BBC.com` times out due to the large number of requests it makes, but most of the content is fetched and rendered (see Fig. 18). We conclude that

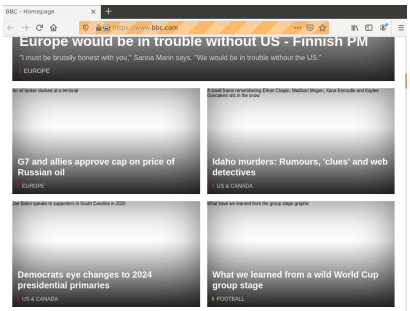


Figure 18: Partially loaded `BBC.com` page. Most of the content is fetched and rendered.

TELEPATH can support the browsing of webpages that do not include a large number of pictures or videos.

Audio streaming. An important use of censorship circumvention systems is to listen to podcasts and other audio content blocked by the censors. To evaluate whether TELEPATH can support audio streaming, we use a Rocket Streaming Audio Server [30] to broadcast audio files with different bitrates, and a VLC media player on the client side that connects to the TELEPATH proxy to listen to broadcast audio. The audio files are in the constant-bitrate MP3 format with bitrates of, respectively, 16Kbps, 32Kbps, 64Kbps, 128Kbps, and 256Kbps. TELEPATH is able to support streaming at all bitrates, except for a very few glitches at the beginning of the playback of the 256Kbps audio file.

8. Related work

Covert communication systems that imitate other network protocols are vulnerable to traffic analysis [15, 40]. Other approaches aim to generate realistic traffic by encod-

ing covert content in audio [16] and video [19] streams, with a significant loss of bandwidth. They can still fail against statistical traffic analysis [1].

There is a large body of work on traffic analysis for website fingerprinting (e.g., [34]) and videostream fingerprinting (e.g., [29, 32]). These techniques identify the specific content being transmitted, as opposed to detecting whether the content was generated by an application or substituted by a circumvention system. For the latter task, we use the traffic analysis methodology of [1, 2].

State-of-the-art censorship circumvention systems like Protozoa [2], Balboa [31], and Camoufler [33] resist traffic analysis by substituting covert content into transport streams generated by an existing application. In Section 4, we explained that in games such as Minecraft, these approaches would generate traces that violate game-specific invariants and would thus be easily recognizable.

Our approach of substituting content only into non-disruptive messages is similar in spirit to Slitheen [3], which substitutes "leaves" of HTML documents to preserve traffic patterns generated by fetched Web content. Slitheen was proposed in the context of decoy routing; our domain, substitution method, and target application are different.

Minecruft [24] is a concurrently developed, yet-unpublished censorship circumvention system based on Minecraft. Unlike TELEPATH, Minecruft encodes covert data into Minecraft packets and inject those packets into the transport layer *in addition to* the Minecraft-generated traffic. It is not clear if this approach resists traffic analysis, since simple features such as packet counts should distinguish Minecruft from Minecraft traces.

Other game-based censorship circumvention systems such as Rook [39] and Castle [13] provide covert communication channels over First Person Shooter (FPS) and Real-Time Strategy (RTS) games, respectively. Both systems have very limited bandwidth and support text data only.

There exist provably secure steganographic techniques for embedding covert messages into text [14, 17, 45]. In steganography, the adversary directly observes cleartexts; the goal is to distinguish their distribution from the distribution of "benign" texts. Provable indistinguishability in this setting is achieved with relatively low encoding capacity, 1-4 bits per lexical token, and a significant computational overhead. Applying this method to non-text channels (such as games) appears non-trivial, since we do not have models capturing conditional content distributions.

By contrast, we work in the same threat model as Protozoa [2] and Balboa [31]. The adversary can only observe a side channel: times and sizes of encrypted packets. Our adversary also knows some application-specific invariants (mandatory orderings of certain events). This side information has no equivalent in provable natural-language steganography. It would require the existence of higher-level invariants satisfied by any benign text (e.g., a mandatory ordering of certain words arbitrarily far from each other in the text) beyond conditional next-token distributions. Unlike for natural text, we do not have generative models that capture realistic traffic distributions for a given application, and even such hypothetical models may only match statistical properties, not order invariants.

9. Conclusions

To the best of our knowledge, TELEPATH is the first system that implements a covert communication channel using a complex, popular application (Minecraft) and produces traffic that does not violate application-specific invariants. We showed that TELEPATH traffic resists statistical traffic analysis and is difficult to distinguish from the traffic generated by a Minecraft farming bot.

Future research directions include (1) increasing the bandwidth of the TELEPATH channel to support interactive and media-rich applications; (2) using machine learning to discover more application-specific invariants over packet traces. Research on traffic analysis and traffic indistinguishability is hampered by the lack of real-world traffic data and models for Minecraft or other online games. Availability of such data could significantly advance the state of the art in covert communication and censorship circumvention.

Decentralized, multi-player games other than Minecraft may have potential for censorship circumvention, too. With transport-level, end-to-end encryption now ubiquitous, games provide high-bandwidth encrypted channels and, critically, they are often popular in countries that practice Internet censorship. Consequently, game-based covert communication systems should be resistant to whitelist filters that block all unknown and unrecognizable traffic.

We view TELEPATH as the first step towards investigating how circumvention can take advantage of popular games while providing robust resistance to traffic analysis. Unfortunately, unlike Minecraft, many games discourage users from modifying clients and servers, but the general principle of non-disruptive content substitution should be broadly applicable to the design of covert, unobservable communication systems.

Acknowledgements. Supported by DARPA and AFRL under Contract FA8750-19-C-0079.

References

[1] D. Barradas, N. Santos, and L. Rodrigues, "Effective detection of multimedia protocol tunneling using machine learning," in *USENIX Security*, 2018.

[2] D. Barradas, N. Santos, L. Rodrigues, and V. Nunes, "Poking a hole in the wall: Efficient censorship-resistant Internet communications by parasitizing on WebRTC," in *CCS*, 2020.

[3] C. Bocovich and I. Goldberg, "Slitheen: Perfectly imitated decoy routing through traffic replacement," in *CCS*, 2016.

[4] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[5] "Fast-paced multiplayer (part II): Client-side prediction and server reconciliation," https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html, accessed: Mar 2023.

[6] "Latency compensating methods in client/server in-game protocol design and optimization," https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization, 2001, accessed: Mar 2023.

[7] C. Connolly, P. Lincoln, I. Mason, and V. Yegneswaran, "TRIST: Circumventing censorship with transcoding-resistant image steganography," in *FOCI*, 2014.

[8] H. Corrigan-Gibbs and B. Ford, "Dissent: Accountable anonymous group messaging," in *CCS*, 2010.

[9] "CurseForge Minecraft Mods," https://www.curseforge.com/minecraft/mc-mods, 2022, accessed: May 2022.

[10] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," in *USENIX Security*, 2004.

[11] R. Ensafi, D. Fifield, P. Winter, N. Feamster, N. Weaver, and V. Paxson, "Examining how the Great Firewall discovers hidden circumvention servers," in *IMC*, 2015.

[12] "Introducing time dilation," https://www.eveonline.com/news/view/introducing-time-dilation-tidi, 2011, accessed: Mar 2023.

[13] B. Hahn, R. Nithyanand, P. Gill, and R. Johnson, "Games without frontiers: Investigating video games as a covert channel," in *EuroS&P*, 2016.

[14] N. Hopper, L. von Ahn, and J. Langford, "Provably secure steganography," *IEEE Transactions on Computers*, vol. 58, no. 5, pp. 662–676, 2008.

[15] A. Houmansadr, C. Brubaker, and V. Shmatikov, "The parrot is dead: Observing unobservable network communications," in *S&P*, 2013.

[16] A. Houmansadr, T. J. Riedl, N. Borisov, and A. C. Singer, "I want my voice to be heard: IP over Voice-over-IP for unobservable censorship circumvention." in *NDSS*, 2013.

[17] G. Kaptchuk, T. M. Jois, M. Green, and A. D. Rubin, "Meteor: Cryptographically secure steganography for realistic distributions," in *CCS*, 2021.

[18] S. Khattak, T. Elahi, L. Simon, C. M. Swanson, S. J. Murdoch, and I. Goldberg, "SOK: Making sense of censorship resistance systems," *PoPETS*, 2016.

[19] R. McPherson, A. Houmansadr, and V. Shmatikov, "Covertcast: Using live streaming to evade internet censorship," *PoPETS*, 2016.

[20] "Minebot," https://github.com/michaelzangl/minebot/tree/forge-1.15.2, 2022, accessed: May 2022.

[21] "Minecraft," https://www.minecraft.net, 2022, accessed: May 2022.

[22] "MinecraftForge," https://github.com/MinecraftForge/MinecraftForget, 2022, accessed: May 2022.

[23] "Minecraft Protocol," https://wiki.vg/Protocol, 2022, accessed: May 2022.

[24] "Minecruft," https://github.com/doudoulong/Minecruft-PT, 2022, accessed: Aug 2022.

[25] H. Mohajeri Moghaddam, B. Li, M. Derakhshani, and I. Goldberg, "SkypeMorph: Protocol obfuscation for Tor bridges," in *CCS*, 2012.

[26] A. A. Niaki, S. Cho, Z. Weinberg, N. P. Hoang, A. Razaghpanah, N. Christin, and P. Gill, "ICLab: A global, longitudinal Internet censorship measurement platform," in *S&P*, 2020.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in Python," *JMLR*, 2011.

[28] "PyAutoGUI," https://pyautogui.readthedocs.io/en/latest/, 2022, accessed: May 2022.

[29] A. Reed and M. Kranch, "Identifying HTTPS-protected Netflix videos in real-time," in *CODASPY*, 2017.

[30] "Rocket Streaming Audio Server," https://www.rocketbroadcaster.com/streaming-audio-server/, 2022, accessed: Aug 2022.

[31] M. B. Rosen, J. Parker, and A. J. Malozemoff, "Balboa: Bobbing and weaving around network censorship," in *USENIX Security*, 2021.

[32] R. Schuster, V. Shmatikov, and E. Tromer, "Beauty and the burst: Remote identification of encrypted video streams," in *USENIX Security*, 2017.

[33] P. K. Sharma, D. Gosain, and S. Chakravarty, "Camoufler: Accessing the censored Web by utilizing instant messaging channels," in *AsiaCCS*, 2021.

[34] P. Sirinam, M. Imani, M. Juarez, and M. Wright, "Deep fingerprinting: Undermining website fingerprinting defenses with deep learning," in *CCS*, 2018.

[35] "tc," https://man7.org/linux/man-pages/man8/tc.8.html, 2022, accessed: Aug 2022.

[36] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, "Stadium: A distributed metadata-private messaging system," in *SOSP*, 2017.

[37] "Property replication in Unreal Engine," https://docs.unrealengine.com/5.1/en-US/property-replication-in-unreal-engine/, 2023, accessed: Mar 2023.

[38] J. Van Den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: Scalable private messaging resistant to traffic analysis," in *SOSP*, 2015.

[39] P. Vines and T. Kohno, "Rook: Using video games as a low-bandwidth censorship resistant communication platform," in *WPES*, 2015.

[40] L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton, "Seeing through network-protocol obfuscation," in *CCS*, 2015.

[41] Z. Weinberg, J. Wang, V. Yegneswaran, L. Briesemeister, S. Cheung, F. Wang, and D. Boneh, "StegoTorus: A camouflage proxy for the Tor anonymity system," in *CCS*, 2012.

[42] P. Winter and S. Lindskog, "How the Great Firewall of China is blocking Tor," in *FOCI*, 2012.

[43] "Wireshark," https://www.wireshark.org/, 2022, accessed: May 2022.

[44] "Xvfb - virtual framebuffer X server for X Version 11," https://www.x.org/releases/X11R7.7/doc/man/man1/Xvfb.1.xhtml, 2022, accessed: May 2022.

[45] S. Zhang, Z. Yang, J. Yang, and Y. Huang, "Provably secure generative linguistic steganography," *ArXiv:2106.02011*, 2021.

Appendix A.
Definition of non-disruptive content

Online game. The game server has the game state $\vec{S} = [s_1\ s_2\ ...\ s_n]$, $\vec{S} \in \mathbb{R}^n$; each s_i is a feature that describes the state. The game client has a set of actions $A = \{a_1, a_2, ..., a_k\}$; an action a_i is a function that takes the game state $\vec{S}$ and generates an update message $u = a_i(\vec{S})$. The server has a function $Apply$ that applies multiple update messages to S and generates a new state $\vec{S'} = Apply(S, u_1, u_2, ...)$.

For each game tick, the server sends the state $\vec{S}$ to all clients. Each client chooses an action $a \in A$ based on the user's input, generates an update message $u = a_i(\vec{S})$, and sends u to the server. The server collects update messages u_i from the clients, applies them to the current state, and generates a new state $Apply(S, u_1, u_2, \ldots)$ for the next tick.

Visual effects. We call the i-th feature in the game state $\vec{S}$ a visual effect if $\forall m \in \mathbb{R},\ \forall \vec{S} = [s_1\ s_2\ \ldots s_n],\ \forall a \in A,\ a(\vec{S}) = a([s_1\ s_2\ \ldots s_{i-1}\ \ m\ \ s_{i+1} \ldots s_n])$, i.e., for a given game state, a client action generates the same update message regardless of the value of this feature in the state.

Synchronization messages. If an action $a \in A$ is chosen by the game client periodically, it is a synchronization action and its corresponding update message is a synchronization message. Since the server periodically sends the game state to clients, the state is a synchronization message, too.