Games Without Frontiers: Investigating Video Games as a Covert Channel

Bridger Hahn, Rishab Nithyanand, Phillipa Gill, and Rob Johnson
Stony Brook University
Email: {bdhahn, rnithyanand, phillipa, rob}@cs.stonybrook.edu

Abstract—**The Internet has become a critical communication infrastructure for citizens to organize protests and express dissatisfaction with their governments. This fact has not gone unnoticed, with governments clamping down on this medium via censorship, and circumvention researchers working to stay one step ahead. In this paper, we explore video games as a new avenue for covert channels. Two features make video games attractive for use as a cover protocol in censorship circumvention tools: First, games within a genre share many common features. Second, there are many different games, each with their own protocols and server infrastructures. These features allow circumvention tool developers to build a single framework that can be adapted to work with many different games within a genre; therefore allowing quick response to censor created blockades. In addition, censored users can diversify their covert communications across many different games, making it difficult for a censor to respond by simply blocking a single covert channel.**

We demonstrate the feasibility of this approach by implementing our circumvention scheme over three real-time strategy games (including two best-selling closed-source games). We evaluate the security of our system prototype, Castle, by quantifying its resilience to a censor-adversary, similarity to real game traffic, and ability to avoid common pitfalls in covert channel design. We use our prototype to demonstrate that our approach can provide the throughput necessary for bootstrapping higher bandwidth channels and also the transfer of textual data, such as web articles, e-mail, SMS messages, and tweets, which are commonly used to organize political actions.

1. Introduction

The Internet has become a critical communication infrastructure for citizens to obtain accurate information, organize political actions [1], and express dissatisfaction with their governments [2]. This fact has not gone unnoticed, with governments clamping down on this medium via censorship [3], [4], [5], surveillance [6] and even large-scale Internet take downs [7], [8], [9]. The situation is only getting worse, with Freedom House reporting 36 of the 65 countries they survey experiencing decreasing levels of Internet freedom between 2013 and 2014 [10].

Researchers have responded by proposing several *look-like-something* censorship circumvention tools. These tools aim to disguise covert traffic as another (benign) protocol to evade detection by censors. This can take two forms: either mimicking the cover protocol using an independent implementation, as in SkypeMorph [11] and StegoTorus [12], or encoding data for transmission via an off-the-shelf implementation of the cover protocol, as in FreeWave [13].

This has created an arms race between censors and circumvention tool developers. For example, Tor's introduction of "pluggable transports", i.e. plugins that embed Tor traffic in a cover protocol to counter censors that block Tor [14]. Censors have already begun blocking some of these transports [15], and some censors have gone so far as to block entire content-distribution networks that are used by some circumvention systems [16].

Furthermore, recent work has shown that care must be taken when designing and implementing a look-like-something covert channel. For example, Houmansadr *et al.* showed that, when a covert channel re-implements its cover protocol, the copy is unlikely to be a perfect mimic of the original protocol, and a censor can use the differences to recognize when a client is using the covert channel [17]. Worse yet, Geddes *et al.* demonstrate that even running the cover application is not enough to avoid detection by censors [18] – i.e., approaches like FreeWave may be detected via architectural, channel, and content mismatches between the application's regular behavior and its behavior when being used as a covert channel.

1.1. The promise of video games

In light of this state of affairs, this paper argues that video games have several features that make them an attractive target for covert channel development.

There are many games available, enabling developers to create a diverse set of circumvention tools. The number of real-time strategy games has grown rapidly in the last few years. This growth has been driven in part by the democratization of game publishing, as embodied in game distribution platforms such as Steam [19] – *e.g.,* Figure 1 shows the total number of real-time strategy video games that have been released since 2010 on the Steam platform. Further, each game uses its own network protocol

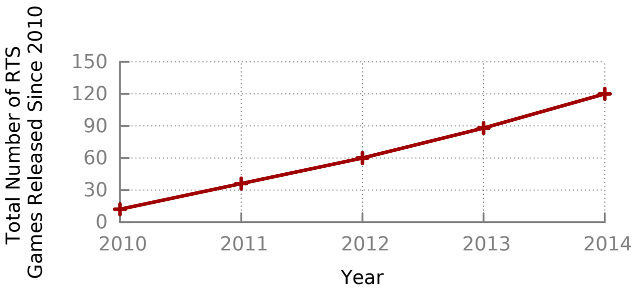


Figure 1: Growth of the real-time strategy game video game genre on the Steam distribution platform [19].

and infrastructure, so the censor cannot simply block all games using a single technique. Censorship circumvention developers can use this large body of games to adapt and evade a censor's attempt to block any particular game.

Video games share common elements, making it possible to use a single framework across many games. For example, most Real-Time Strategy (RTS) games have the notions of buildings, units, and rally points, and censorship circumvention tools that encode information by interacting with these objects can be easily ported from one RTS game to another. Many games also feature replay logs and similar user interfaces, enabling covert channel frameworks that are only loosely coupled to the internals of any particular game.

Game-based circumvention tools can re-use off-the-shelf game implementations Since games have features that make it relatively easy to automate interaction with the game, circumvention tool developers do not need to re-implement the game (or its network protocol), ensuring that the circumvention tool can leverage the existing implementation of the game. This prevents attacks that can distinguish between the original implementation and the cover-protocol implementation of an application or protocol [17].

Game-based circumvention tools avoid previously encountered pitfalls. Games in select genres often support both peer-to-peer and server-based gaming sessions (*e.g.,* real-time strategy games), so they can adapt to whichever is better for the circumvention tool. This allows architectural matching as described by Geddes *et al.* [18]. Games must maintain synchronized state, so they are loss sensitive, avoiding the channel mismatch between multimedia and Web/textual covert content identified by Geddes *et al.* [18]. Finally, games are reasonably able to avoid content mismatches by due to the large amount of diversity in typical content characteristics.

Games often have built-in security features that can support secure covert channels. It is considered good practice for games support encryption and authentication in order to prevent cheating [20], [21] – *e.g.,* the Microsoft DirectX networking API [22] and the Steam peer-to-peer networking API [23] which are commonly used by game developers include support for SSL sockets. Additionally, some games also support password-protected sessions, which can prevent application-level attacks in which the censor attempts to identify covert channels by joining the game.

Games have the potential to reverse the resource imbalance in the arms race between censors and developers. By lowering the development cost of creating new covert channels, video games can create an asymmetry that circumventors can use to win the arms race against censors. Censors can respond to look-like-something circumvention tools by blocking the cover protocol entirely or attempting to distinguish legitimate uses of the protocol from uses by the covert channel. If developing such mechanisms is time consuming for the censor, but circumvention tool developers can quickly construct new tools, there will almost always be effective circumvention tools available for end users.

1.2. Our contributions

In spite of the above benefits, we must answer several questions to understand the feasibility of using video games for covert channels:

- **Security:** Can we encode data in the video game so that the censor cannot distinguish regular game play from covert channel sessions?
- **Extensibility:** Can we build a framework that can be quickly adapted to new games?
- **Performance:** Can video games support good covert channel bandwidth?

To answer these questions, we have built Castle, a prototype video game-based covert-channel framework. Castle encodes data as player actions in an RTS game. Castle uses desktop-automation software to execute these actions in the game. The video game software transmits these moves to the other players in the same gaming session, who then decode the message and send replies in the same way.

Security. Castle's design makes it resilient to several classes of attacks. Since Castle uses the underlying game to transmit data, an attacker cannot use simple IP- or port-based blocking to block Castle without blocking the game entirely. When used with games that encrypt and authenticate their traffic, an attacker cannot use deep packet inspection to distinguish Castle traffic from regular game traffic. Encryption and authentication also preclude simple packet injection or manipulation attacks. Since games use network communication to synchronize their state, they are loss sensitive, unlike some VoIP protocols. Thus Castle cannot be distinguished from regular gaming sessions through selective packet delay or dropping attacks. Finally, when used with password-protected gaming sessions, Castle is immune to application-level attacks, such as the censor attempting to join the same gaming session to observe the player's in-game actions.

We evaluate Castle's security against statistical traffic-analysis attacks by applying several previously published classifiers – *i.e.,* the Liberatore [24], Herrmann [25], and Shmatikov [26] classifiers. We find that packet sizes and inter-packet times of Castle's traffic deviate from those of regular human-driven game play by the same amount that different human player's traffic differ from each other.

Extensibility. Castle can be easily adapted to new RTS games. Our current prototype supports three such games: "0-A.D." [27] and two extremely popular (over 8.5 million copies sold) closed-source games from different development studios that we refer to as "Aeons" and "Conquerors". It took a single undergrad less than six hours to port Castle from 0-A.D. to each game.

Castle is easy to port to new RTS games for two reasons. First, Castle uses only features that are nearly universal to RTS Games – *e.g.,* gameplay characteristics and game replay features. Thus the high-level architecture and encoding scheme can be re-used across games. Second, Castle is only loosely coupled to game internals – requiring no access to the game source-code. For example, Castle uses desktop-automation software to execute game actions through the game's standard graphical user interface. As a result, Castle does not need to understand the game's network protocol or any other internals.

Performance. Castle offers good bandwidth for text-based communications. Our current prototype provides between 50 and 200 B/s of bandwidth, depending on configuration parameters. Castle has about 100x more bandwidth than other proposed game-based covert channels [28], [29], [30][1]. With some game-specific tuning, the Aeons version can deliver over 400 B/s. Even 50 B/s is sufficient for bootstrapping high bandwidth communication channels (e.g., distributing Tor bridge IPs), text-based web articles, email, SMS messages, tweets, and other asynchronous communications which are widely used organizational tools among political activists. There are also several ways to potentially increase Castle's bandwidth (see Section 8 for details).

Together, these results show that video games offer promise as a target for covert channel development and they may enable circumvention tool developers to gain the upper hand in the arms race against censors.

Paper outline. In Section 2, we present the adversary model that we consider in this paper. Section 3 provides background on real-time strategy games, details the properties that makes them favorable for use as cover protocols in covert channels, and explains how Castle makes use of each of these for sending and receiving covert data. In Section 4, we provide details on our publicly available implementation of Castle. Following this, we describe our evaluation criteria in Section 5. In Sections 6, 7, and 8, we present the results of Castle's security, extensibility, and performance evaluation, respectively. In Section 9, we compare the primary design principles of Castle with its most similar counter-parts. Finally, in Section 10, we draw our conclusions.

2. Adversary and Threat Model

In this paper, we consider a network-level censor (e.g., an ISP) able to (1) perform analysis over all traffic that it forwards from or to clients within its network and (2) perform manipulations (*e.g.,* dropping and injecting packets) of the network traffic via on-path or in-path middleboxes. In addition, the adversary may also take an active approach by probing and interacting with application endpoints.

2.1. Network traffic attacks

Passive analysis. We consider censors that are able to perform stateless and stateful passive analysis of traffic at line rate. In particular, the censor is able to perform the following passive analyses to detect the use of a circumvention tool:

- **IP and port filtering:** The censor can observe the IP addresses and port numbers of connections on their network (*e.g.,* using tools like Netflow [31]).
- **Deep-packet inspection:** The censor may look for specific patterns in packet headers and payloads (*e.g.,* payloads indicative of a specific game).
- **Flow-level analysis:** The censor may perform statistical analyses of flow-level characteristics such as inter-packet times and sizes) while maintaining a reasonable amount of state.

The first two of these capabilities mean that the ISP can detect flows related to the video game in general. For example, if the game uses a specific set of servers (IPs) or ports, these flows may be easily identified. Similarly, game-specific payloads can reveal game traffic to the ISP. The last property can reveal information about game behavior to the ISP. A circumvention system must avoid perturbing these features to remain undetected and unblocked.

Active manipulations. In order to detect and/or disrupt the use of censorship circumvention tools, censors may perform a variety of active manipulations on suspicious connections that transit its network. In particular, the censor may drop, insert, or delay packets. Additionally, they may also modify the packet contents and headers. The adversary may perform these manipulations to observe the behavior of flow endpoints to distinguish legitimate game traffic from the covert channel. They may also use these actions to block covert connections (*e.g.,* sending TCP RST packets, or dropping traffic).

2.2. Application layer attacks

In the context of detecting *look-like-something* covert channels, censors may take additional actions outside the scope of standard active and passive analysis. Specifically, they may interact with the application that the covert channel aims to hide within. They may attempt to join game servers and observe games in progress (*i.e.,* to identify who is playing with whom). Additionally, they may seek to observe properties of the games being played (*e.g.,* map state, player move behaviors) or join and interact with game players.

2.3. Censor limitations

We impose limitations on the computational and storage capabilities of censors. While they have a large amount of

[1] Despite the similarity of their names and their common use of video games, Rook and Castle were developed independently and have quite different goals. See Section 9 for details.

computational resources, they are still unable to maintain a large amount of per-connection state for long durations or decrypt encrypted communication channels and guess high entropy passwords. We also assume that the censor does not have a back door into the game or game servers. For example, we assume the censor is not able to break into the game servers (e.g. by exploiting a buffer overflow or other bug). We also assume that the operators of the game servers do not cooperate with the censor, e.g. they do not allow the censor to see other user's private game state.

3. The Castle Circumvention Scheme

Castle aims to demonstrate that highly portable, secure, and low-bandwidth *look-like-something* defenses are possible via applications such as real-time strategy video games. In this section, we provide a background on the real-time strategy genre and highlight key properties of these games that enable Castle to create covert channels that generalize to a large number of games within the genre. Finally, we describe how Castle encodes, sends, and receives data.

3.1. Real-time strategy games

Real-time strategy games are a genre of video games that center around the idea of empire-building. Typically, the goal is for a player to assert control over enemy territory through a combination of military conquest and economic maneuvering. Below we highlight commands and features that are common to a large majority of real-time strategy games (Table 1) and are critical to the extensibility of Castle.

Units. Real-time strategy games allow players to *create* and *train* a large number of units (*e.g.,* human characters, livestock, machinery). Units may perform many actions. For example, in 17 of the Top 20 best-selling real-time strategy games [32], a unit can be instructed to move to a location on the map by left-clicking it and then right clicking the destination location on the map.

Buildings. Players may construct a number of buildings over the course of a game. Buildings are required to train certain units and research new technologies – *e.g.,* barracks are required to train infantry. In many (*e.g.,* in 17 of the Top 20 best-selling) real-time strategy games, unit-producing buildings can be assigned a rally point – *i.e.,* a location at which all units created by the building will assemble.

Maps and map editors. Real-time strategy games are set in a landscape covered by plains, forests, mountains, and/or oceans. Many (including 17 of the Top 20 best-selling) real-time strategy games allow users to create and use their own maps, or modify existing maps for use within the game. This is either from within the game, or via external mods.

Replay files. In newer games, players may be given the option to record all moves and commands issued by themselves and other players in the game. This is used to replay or watch previously played video games. When this option is enabled, the game writes, in real-time, all in-game commands to a replay log. While this replay log may be stored in a propietary format, we found decoders to read these formats are available for 11 of the Top 20 best-selling real-time strategy games.

Feature	Number of Games
Common Comands	17
Map Editors	17
Replay Decoders	11

TABLE 1: Real-time strategy game features used by Castle and the number of games in the Top 20 best-sellers of all-time that possess them [32].

In addition to the above elements, the following networking and security properties are standard in the real-time strategy genre. These properties make real-time strategy games very suitable for use as covert channels.

Network communications. For scalability reasons, real-time strategy games do not broadcast state information to all players in the game. Instead, they pass commands issued by the players in fixed intervals (*e.g.,* 100 ms). These commands are then simultaneously simulated in each game client. This allows clients to execute the game identically, while requiring little bandwidth [33]. As a consequence, any data encoded as an in-game command is received as such, by other players.

Additionally, while most real-time strategy games make use of UDP channels for command communication, reliable delivery mechanisms are implemented in the application layer. This makes many active traffic manipulation attacks described in previous work [18] ineffective.

In terms of network architecture, real-time strategy games may take two forms, with players joining a game hosted on a public game server (*e.g.,* servers hosted by game publishers such as Microsoft, Blizzard, Electronic Arts, etc.), or connecting directly to each other in a peer-to-peer mode. Therefore, any covert channel system utilizing video games as a cover, can employ whichever is the dominant mode of operation and shift from one to the other if required, to evade a censor blockade.

Security considerations. In order to prevent cheating in the multi-player setting, it is considered good practice to implement encrypted and authenticated communication channels, in real-time strategy games [20], [21]. Additional mechanisms such as verification of game-state consistency (between all clients in the game) [33] and password-protected multi-player game sessions [23] are also common.

These security mechanisms have several vital consequences in the context of using real-time strategy games as covert channels. First, since the game command channel is encrypted, passive adversaries are unable to view commands issued by players in a game by simply observing network traffic. Second, the presence of authenticated channels and game-state verification algorithms prevents active attackers from using falsified game packets to interact with, or observe other clients on the game servers. Finally, the presence of password-protected game instances prevents adversaries from joining multi-player games (to observe the in-game state and identify players).

Commonalities between real-time strategy games. Our design leverages the common command structure, map design capabilities, and tools for decoding saved games and replays generated by real-time strategy games. Table 1 shows the results of our survey of the prevalence of these features in real-time strategy games. We find that 11 of the top 20 best-selling games of all-time include these features.

3.2. Building game-based covert channels

Straw-man approach. One may consider establishing covert channels via the in-game voice or text chat channels. However, this approach has several drawbacks.

First, previous work shows that encoded data is easily distinguishable from human audio communication [17], [18]. Furthermore, voice communication channels are fairly uncommon in the real-time strategy game genre. Second, while game data is encrypted, it is often the case that text communication channels are left unencrypted. Finally, while one may expect a fairly constant stream of human issued in-game commands in a real-time strategy game, it is rare to have long text or audio communication while playing the game. These factors allow covert channels built on these approaches to be either difficult to implement/extend, or to be easily detected by an adversary, or both.

The Castle approach. To create a covert channel mechanism that is extensible to a large number of games in the real-time strategy genre, Castle exploits two key properties.

- **Presence of common commands.** Real-time strategy games share a common set of commands. Specifically, the ability to select buildings and assign a location where units created in these building should assemble. This location is called a "rally point". We denote the command of setting the rally point for units created in a given building by `SET-RALLY-POINT`. Games also provide the ability to move a selected unit to a given location (denoted by the `MOVE` command). Thus, any encoding that translates covert data into a combination of unit/building selections and these primitives will be extensible across games in the genre.
- **Access to replay logs.** Often, real-time strategy games provide a replay option which saves every players' moves to disk (for later playback). Therefore, all in-game commands are written to disk where they can be read and decoded in real-time.

Castle consists of two main components to send and receive data. These are illustrated in Figure 2. Sending is done by encoding data into game commands and then executing them within the game using desktop automation. The receiving process monitors the log of game commands and decodes this list to retrieve data sent via the system.

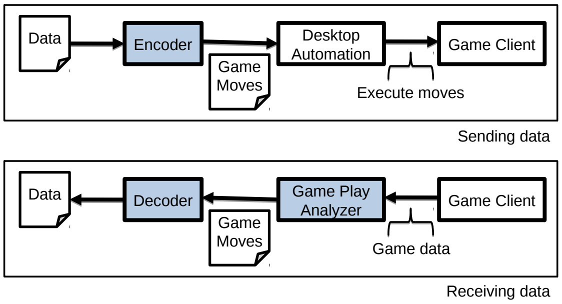


Figure 2: Overview of data flow for sending an receiving in Castle. Shaded components are implemented as part of Castle while the others use existing off-the-shelf software.

3.3. Encoding data into game commands

Castle relies on the ability of the player to select units and buildings and set rally points to encode data. A naive encoding may consider selecting each unit and directing it to a different point on the game map to encode a few bytes of information per unit. However, in preliminary experiments, we observed that this approach resulted in a covert channel that could not match the properties of the original game traffic (moving O(100s) of units to distinct locations is not a usual action for players).

Encoding in Castle is accomplished, without inflating the amount of game data transferred, using the following scheme. First, the participants in Castle use (standard or Castle-specific) maps which contain either n immobilized units (e.g., units placed in unit sized islands, within walls, etc.) or n unit producing buildings (e.g., barracks, stable, etc.). The Castle sending process then encodes data by selecting a subset of these n units and executing either a `MOVE` command in the case of units or `SET-RALLY-POINT` in the case of buildings. While we discuss the encoding in the context of units and the `MOVE` command, Castle is easily implemented using either primitive.

Instead of using each of the n units to represent a single bit sequence, which would result in $\log_2(n)$ bits of data transferred per command, we use a combinatorial scheme where we select k of the n units, to increase efficiency. Intuitively, the selection of k of n units results in $\binom{n}{k}$ different values or $\log_2\binom{n}{k}$ bits that may be transferred per command. We use combinatorial number systems [34] to convert $\log_2\binom{n}{k}$ bits of data into a selection of k of the n units. In preliminary experiments, we found that the selection of a constant number of units per command resulted in traffic which was more uniform than regular game traffic. As a result, we adjusted our scheme to select between 0 and k units for encoding to increase variability of packet sizes. Section 6 provides a more in-depth view of how we evaluate our similarity to actual game traffic.

In addition to selecting the set of units, we can also select a location for all k selected units to move to. Note that since we select a single location for k units (instead of k distinct locations) this does not impact the data transfer size. Given a game map with $m = x_{max} \times y_{max}$ potential locations we can additionally encode $\log_2 m$ additional bits of data in a given turn.

Assuming a map with n units/buildings, a maximum of $m = x_{max} \times y_{max}$ map locations, and a game which allows for a maximum of k units/buildings to be selected simultaneously, the game-independent encoding of covert data into a `MOVE` or `SET-RALLY-POINT` command is done as shown in Algorithm 1.

Algorithm 1 Algorithm for encoding covert data into game commands

```
function ENCODE($data$, $k$, $n$, $m$, $x_{max}$)
    $r \xleftarrow{\$} \{1, \ldots, k\}$
    $z_1 \leftarrow$ READ($data$, $\log_2 \binom{n}{r}$)
    for $i = n \rightarrow 0$ do
        if $\binom{i}{r} \leq z_1$ then
            $z_1 \leftarrow z_1 - \binom{i}{r--}$
            $selected \leftarrow selected || i$
        end if
    end for
    $z_2 \leftarrow$ READ($data$, $\log_2 m$)
    (x, y) $\leftarrow (z_2 \mod x_{max}, \lfloor z_2 / x_{max} \rfloor)$
    return $\{selected$, (x, y)$\}$
end function
function READ($file$, $b$)
    return next $b$ bits from $file$ in base 10.
end function
```

The combination of selecting between 0 and k units and setting the location to move to, results in an average of $\left(\frac{\sum_{i=1}^{k} \log_2 \binom{n}{i}}{k} + \log_2 m \right)$ bits transferred per command.

As mentioned earlier, one may achieve higher data-rates by always selecting k units, however, this causes identically sized commands and thus affects the packet size distribution.

3.4. Sending covert data

Once the covert data is encoded into in-game commands, the sending process must actually execute the commands in order to communicate them to the receiver. One way to do this is to modify the game AI to issue commands as dictated by our encoder. However, this is non-trivial since most games are closed-source and viewing/modifying game code is not always an option. Even when source code is available, the overhead of understanding the game code and modifying the AI presents a non-trivial hurdle. Given our vision of adaptability to the large number of available real-time strategy games, we leverage off-the-shelf desktop automation to execute the encoded game commands. This opens the door to extending our approach to a larger set of games than would otherwise be possible.

Since the map used in Castle is custom made, the starting location of all units is known in advance. Further, since units and buildings are immobile, Castle is aware of their location at all times. The location of units on the game map, along with the list of commands to be executed is sufficient for Castle to automatically generate a sequence of key-presses, and clicks to be made by the desktop automation tool. This sequence is then passed to the automation tool for execution.

We note that, certain automation tools allow keystrokes and clicks to be sent to windows that are not currently in focus. This ensures that Castle does not detract from the user experience by requiring the game window to be in focus during data transfer periods. Finally, since automation tools allow control over the speed of clicks and key-presses, Castle can be configured to either mimic human input speeds (lower clicks/second) or maximize throughput (higher clicks/second). We investigate the trade-off between these two variables in Sections 6 and 8.

3.5. Receiving covert data

Since the receiving game client does not have the same in-game screen as the sending client (due to each client having their camera focused on different map locations), directly observing the commands made by the sending client via the screen output is prohibitively complex. Fortunately, most real-time strategy games maintain a real-time log of all commands issued in the game to enable replaying moves or saving game state. In Castle, the receiving process constantly monitors this log file for commands issued by other participants. These commands can then be decoded back into their original covert data via the decoding algorithm specified in Algorithm 2.

Algorithm 2 Algorithm for obtaining covert data from game commands

```
function DECODE($selected$, (x, y), $x_{max}$)
    $size \leftarrow |selected|, z_1 = 0$
    $selected \leftarrow$ SORT-DESCENDING($selected$)
    for $i \in selected$ do
        $z_1 \leftarrow z_1 + \binom{i}{size--}$
    end for
    $z_2 \leftarrow$ (y $\times x_{max}$) + x
    return $(base2(z_1) || base2(z_2))$
end function
```

This approach suffers from one minor drawback: replay logs for games from commercial studios are often stored in proprietary and undocumented formats that vary from game to game. However, reverse engineering the format of the replay logs is made significantly easier since Castle only issues `MOVE` or `SET-RALLY-POINT` commands. Therefore, we only need to understand how these commands are stored in replay logs. This can be done by simple techniques – e.g., sending a unit to the exact same location multiple times allows us to obtain the byte code used to signify the `MOVE` command, sending a unit to two locations in sequence (with each separated by a single pixel) allows us to obtain the bytes used to denote the `(x, y)` destination coordinates, etc. Further, for many popular real-time strategy games, these formats have already been reverse-engineered by the gaming/hacking community – *e.g.,* 11 of the Top 20 real-time strategy games have decoded replay file formats.

Additionally, it is important to note that: (1) The overhead of decoding replay files is amortized over the entire set of users using that game as a cover, and (2) It is common for replay formats to be identical for real-time strategy games published by the same studio – *e.g.,* most Microsoft real-time strategy games use the MGX replay format. Therefore, a working decoder for one game from one particular studio may work for all games from the same studio.

3.6. Bootstrapping Castle communication

In order to bootstrap covert communication, the following information needs to be shared between Castle users:

Castle user identity and configuration. For a covert channel to be established, a Castle user must first be able to find and join Castle game instances. Due to the absence of pre-established secrets, doing this in a secure way (*i.e.,* a way that cannot also be used by the adversary censor to identify Castle game instances) is a currently an open research problem. We envision that current solutions such as BridgeDB [35] can be used for distributing game instance identities and configurations. In particular, BridgeDB may be used to distribute names and passwords of Castle game instances (in the case of games hosted on public servers) and IP address/port numbers of Castle games (in the case of peer-to-peer *direct-connect* game instances).

Castle map. In order to establish a covert communication channel, Castle users may also need to share a common Castle compatible map (that is used by the Castle game instance). While such maps might be quite large (in the order of a few MB), Castle provides a generic map generation script that is able to generate identical maps for all clients in the game with just a few bytes of configuration information.

Generally, to automate the process of Castle compatible map creation via a map editor, one needs a subroutine for creating buildings at specific locations on the map. Given this single subroutine, it is possible to automate the entire map generation process. In many map editors (*e.g.,* map editors of 17 of the Top 20 best-selling real-time strategy games), we observe that such a subroutine only requires the automation of two clicks – one on a button to select the building type and one on the location at which the building is to be placed. For such editors we provide a generic Castle map creation automation script which only requires the following information for its building placement subroutine: the location of the button for the desired building type, the dimensions of the selected building type, and the available screen space. This information requires only a few bytes and allows users supplying the same parameters to generate identical maps. Additionally, it can be shared using the identity and configuration distribution mechanism.

4. Castle Prototype Implementation

In this section, we describe our implementation of Castle. We prototype on three games, each from a different publisher, to illustrate the extensibility of our approach.

- **0 A.D.:** An award-winning, free, open-source, and cross-platform real-time strategy game made available under the GPLv2+ license, by Wildfire Games.
- **Aeons:** A best-selling (in the top 2 grossing real-time strategy games of all-time), closed-source, Windows-based real-time strategy game from Studio X.
- **Conquerors:** A best-selling (in the top 5 grossing real-time strategy games of all-time) closed-source, Windows-based real-time strategy game from Studio Y.

Our prototype comprises of $\sim$500 LOC and was coded in a combination of Python and AutoHotkey (desktop automation) [36] scripts. It includes the following components:

Custom map. To test Castle, we created a custom game map for each of the three games. The map was comprised of n buildings packed as tightly as possible to facilitate our selection-based encoding. For 0 A.D., we created a map with $n = 1600$ buildings on a single game screen, while for Aeons and Conquerors, we were only able to have $n = 435$ and $n = 416$, respectively (owing to larger unit sizes). For all games, a region large enough to contain 16 bits of location data was left unoccupied. This is used to assign rally-point coordinates to the selected buildings.

Since 0 A.D. stores maps in a simple and readable XML format, the process of map creation was easily automated (via a Python script). This was not the case for Aeons and Conquerors which required manual generation of the map using the official GUI map editor. However, the current version of Castle comes with an easy to configure automation script to automatically generate Castle maps for many real-time strategy game requiring map generation via GUI editors (including Aeons and Conquerors).

Data encoding and decoding. Code for translating between covert data and in-game commands (and vice-versa) was written in under 200 lines of Python using the encoding and decoding described in Section 3.3. The output of the encoding code was a vector of buildings to be selected and a single `(x, y)` coordinate.

Desktop automation. We used the open-source desktop automation tool, AutoHotkey, to execute the series of commands output by the encoding scheme. Since custom maps were used, the location of all buildings and units were known. As a result, selecting and commanding those indicated by our encoding program was straightforward.

Reading recorded game data. We implemented code that monitored the log file of commands issued (maintained by the game), for all games. For 0 A.D., this information was already made available in a simple to parse text file. In order to obtain this information for Aeons and Conquerors, the game replay file was parsed using replay-decoder tools and information made available by the gaming/hacking community. The file was then scanned to obtain each command as a vector of selected buildings and an `(x, y)` coordinate.

The commands were then decoded to retrieve the originally encoded covert data.

Coordinate calibration. The isometric perspective of the game screen posed a challenge during the decoding process. Specifically, the presence of a *viewing angle* meant that a sender may have intended to move a unit to the screen coordinate `(`x_s`, `y_s`)`, but the game actually logged the command as an order to move the unit to the game coordinate `(`x_g`, `y_g`)`, making this the command obtained by the receiver on decoding the move log. To avoid this, Castle goes through a one-time calibration process of mapping on-screen coordinates to coordinates as interpreted in the game. Note that the results of this calibration process can be shared across game clients that have the same resolution.

5. Evaluation Setup

We evaluate Castle along three axes – security, extensibility, and performance. In Section 6 we consider security of the Castle by quantifying its resilience to the censor-adversary described in Section 2 and its ability to avoid the mismatches highlighted by Geddes *et al.* [18]. Next, in Section 7 we evaluate the extensibility of Castle– i.e., how easy is it to implement Castle over a closed-source game. Finally, in Section 8 we study throughput of Castle using the encoding scheme laid out in Section 3.

For the evaluation in Sections 6 and 8, we use our implementation of Castle with a building-based map, using `SET-RALLY-POINT` commands.The evaluation was performed on Windows 8.1 running AutoHotkey [36] for automation. The game was set up in direct connect mode – i.e., the two players were connected directly to each other via their IP address (rather than through the game lobby). Since both players were on the same (fast) university network, negligible effects of lag were experienced.

Castle was used to transfer a randomly generated (via `/dev/urandom`) 100KB binary file from one player to another. Network traffic generated by the game was captured using *Rawcap* (a command-line raw socket packet sniffer for Windows) with additional processing done using `tcpdump`.

We considered the impact of command rate (*i.e.,* how long AutoHotkey waits between each issued command) and the impact of the maximum number of buildings selected (k) on the performance and security of Castle. For this we varied the command delays from 100 to 1000 ms/command and the number of selected buildings from 25 to 200.

In order to compare the traffic characteristics of Castle with characteristics of the standard game, we gathered network traces of regular 0 A.D. two-player games. These were also collected in a similar setting – i.e., with both players on the same university network and via direct connect. Ten traces were collected (one per game played). Each of the recorded games was between 20 and 60 minutes long.

In order to evaluate the extensibility of Castle, armed with a working implementation of Castle over 0-A.D., an undergraduate researcher was given the task of implementing Castle over the popular closed-source Aeons and Conquerors. Finally, to observe the impact of game-specific modifications, we evaluated the throughput of Castle over 0-A.D, Aeons, and Conquerors with and without any game-specific modifications, in the same settings described above.

6. Security Evaluation

We now perform an evaluation of Castle against the network adversary described in Section 2.

6.1. Resilience to network traffic attacks

Passive analysis. We first consider attackers with the ability to perform IP and port filtering, deep-packet inspection, and simple flow-level statistical analysis at line rate.

IP and port filtering: Since Castle actually uses an off-the-shelf implementation of the game application, the IP address and ports used by Castle are identical to that of the standard use of the game. This means that an adversary that triggers blocking based on the destination IP (e.g., the game server) or port number, will be forced to block all traffic to and from the game being used as the cover protocol.

In the event that the censor is willing to block *all connections* to dedicated game servers (often hosted by game publishers – e.g., Electronic Arts, Microsoft, Blizzard, etc.), clients may still utilize Castle in direct-connect (*peer-to-peer*) mode, forcing the censor into a game of whack-a-mole with Castle proxies hosted outside their jurisdiction. Further, users may also easily migrate Castle to another real-time strategy game whose game servers are unblocked.

It is also worth noting that blocking game flows is not without any costs to the censor, specifically with respect to political good will and PR internationally. For example, blocking all traffic for a given game, especially a popular title, may upset citizens and reflect poorly on Internet freedom within the censoring country [37], [38], [39].

Deep-packet inspection (DPI): When used with games that encrypt their communications, Castle is resistant to deep-packet inspection, since the censor cannot decrypt the stream of moves being made. However, since Castle works by issuing only generic commands (*i.e.,* `MOVE` and `SET-RALLY-POINT` commands), it can easily be detected by DPI boxes if the game communicates commands in plaintext. Fortunately, it is generally recommended that real-time strategy games perform command channel encryption, making them resilient to DPI [20], [21], [22], [23].

Flow-level statistical analysis: To quantify the resilience of Castle against flow-level attacks, several statistical tests and classifiers were employed. For each experiment, the Castle parameters that control the command rate and the maximum number of buildings selected were varied between 0 to 1000 ms and 25 to 200 buildings, respectively.

First, the *Kolmogorov-Smirnov* (KS) statistic was used to compare the similarity of human-game-generated traffic and Castle-generated traffic. Figure 3 reflects the KS similarity statistic on the packet size distributions of human- and Castle-generated games and Figure 4 does the same for inter-packet times. We make two observations from

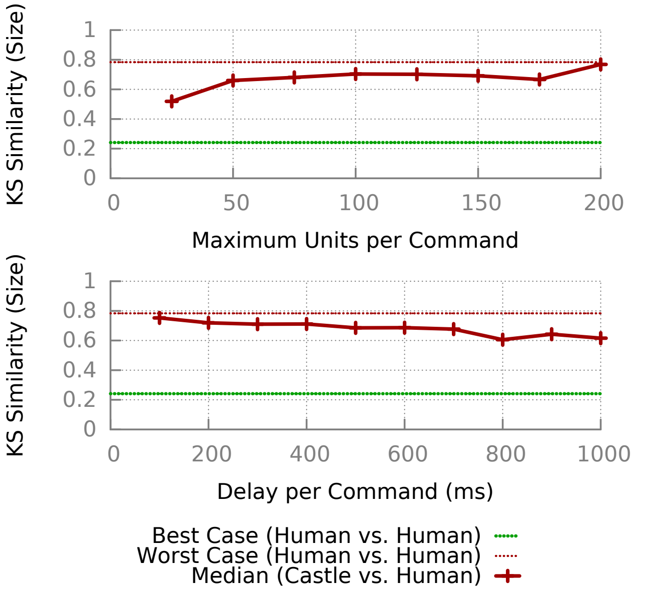


Figure 3: Kolmogorov-Smirnov (KS) statistic on the distributions of packet sizes. The difference between Castle and the legitimate game flows is within the variance observed when comparing traffic between legitimate game flows.

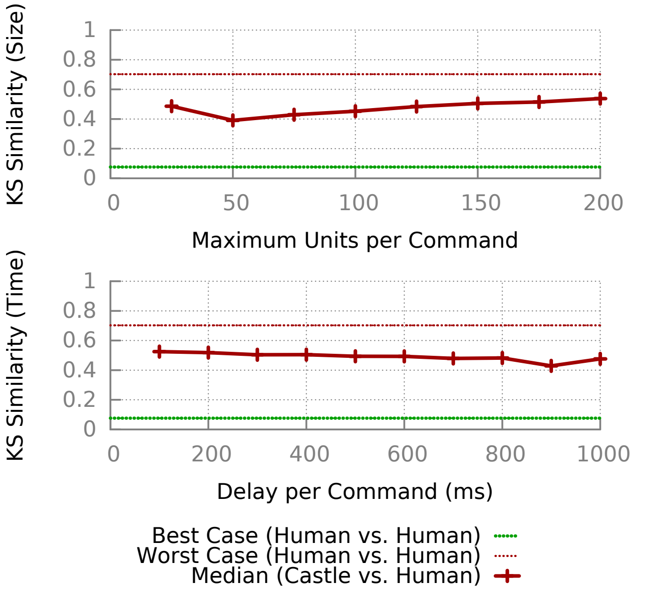


Figure 4: KS statistic on the distributions of inter-packet times. The difference between Castle and the legitimate game flows is within the variance observed when comparing traffic between legitimate game flows.

these plots: (1) There is a high variation in the flow-level features of legitimate (i.e., human-game-generated) traffic. We hypothesize that this is because the traffic generated by the real-time strategy game is strongly dependent on many parameters such as map and scenario type, strategies employed, and number of players. (2) Castle in many configurations, generates traffic that is well within this variance. We find that while restricting the maximum number of units per command to under 50 and the command rate to around 1 command/second, Castle generates traffic that is as similar to traffic generated by legitimate games.

Next, Castle was evaluated against several traffic fingerprinting classifiers. The goal was to evaluate the accuracy of classifiers, built for flow-level analysis, in distinguishing between Castle-generated and human-generated traffic.

First, each network capture was split into (20) one minute long chunks. For each experiment, classifiers were given 20 chunks of Castle-generated traffic at a specific configuration and 20 randomly selected human-game-generated chunks. Ten-fold cross validation was employed for splitting into training and testing sets.

Since, in our experiments, Castle was used for the purpose of file transfer, all traffic generated by it was in a single direction. This makes it trivially detectable by some fingerprinting classifiers which are heavily reliant on burst and direction features (e.g., k-NN [40], the Panchenko classifier [41] and OSAD [42]). We note that in a real deployment this directionality would not be an issue as there would be requests and responses from both sides.

Due to the directionality of traffic, traffic fingerprinting classifiers that ignored directional information were used. These included the Liberatore classifier [24], the Herrmann classifier [25], and an inter-packet timing classifier [26]. All classifier implementations were obtained from Wang's open-source classifier archive [43]. The results of these experiments are illustrated in Figure 5. In general, the results indicate that Castle performs very well against packet size and timing classifiers, with only the Herrmann classifier achieving an accuracy of over 60% against multiple configurations of Castle– *i.e.,* only the Herrmann classifier achieved 10% higher accuracy than random guessing.

Active traffic manipulations. In the face of active traffic manipulation attacks, such as probing, packet injection, and modification, Castle implemented over most commercial games faces little threat.

Packet injection. If Castle is implemented over a real-time strategy game with an encrypted and authenticated command channel, any packets injected by an unauthenticated source are dropped by the game-server. As a result, a probing adversary learns nothing about the Castle games running on the server.

Packet modifications. Most packet modification attacks are prevented by the presence of encrypted and authenticated in-game channels. Additionally, since Castle does not require any changes to the game or the hosting server, such attacks will always elicit the same response from both, legitimate game players and Castle users.

Packet dropping and delaying. Although most com-

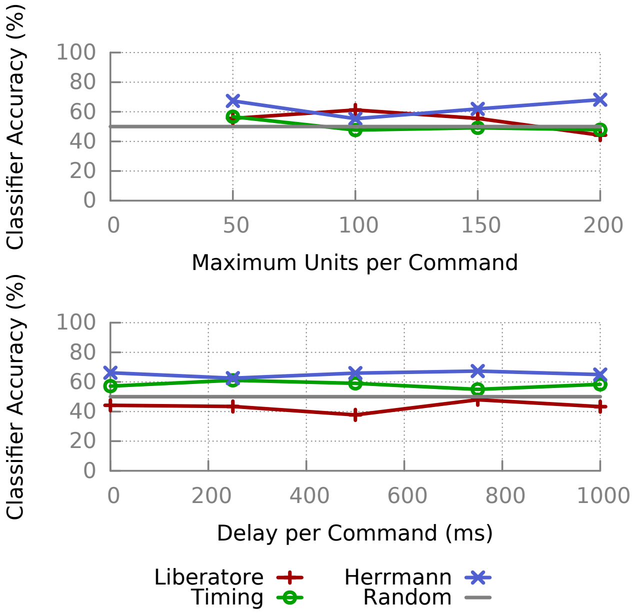


Figure 5: The performance of Castle in various configurations against website fingerprinting classifiers.

mercial real-time strategy games make use of UDP as a transport, the presence of reliability implemented in the application layer prevents any threats from adversaries that drop, or significantly delay packets in transmission. As a result, attacks (*e.g.,* [18]) that result in denial-of-service for Castle users are not possible without also affecting legitimate game players.

6.2. Resilience to application layer attacks

Highly motivated censors may perform actions outside the realm of standard network traffic analysis and manipulation. We consider censors that may attempt to interact with the game server using custom game clients in order to reveal the identities of Castle users. Specifically, censors may connect to game server lobbies to identify Castle games and try to join these games to learn the IPs of participating clients. For these cases, Castle provides several defenses based on features available in the game.

If the cover game supports the use of password-protected multi-player games, Castle proxies (*i.e.,* hosts of Castle games) may configure Castle game instances to require users to authenticate using high-entropy passwords distributed using, for example, the BridgeDB mechanism [35]. Therefore censors without knowledge of the password are unable to join hosted games and learn the IP addresses of Castle users.

If the cover does not support the use of password-protected games, a Castle proxy may incorporate either (or, both) of the following defenses against these adversaries: (1) The proxy may use standard game maps rather than custom-made Castle game maps. This allows Castle instances to blend in with legitimate game instances, making it harder for the censor to identify which games to join. However, this comes at the cost of lower throughput since there are typically fewer units in standard game maps. (2) The proxy may still use a BridgeDB-like mechanism for password distribution and require that any Castle client makes the moves corresponding to the supplied password in order to receive proxying services. In the event that a client does not supply this password within some period of time, the Castle proxy may continue playing the game using a standard AI. Therefore, even a censor that may enter games is unable to distinguish between Castle games and legitimate games.

Deniability and ease of distribution. In addition to being resilient to computational attacks, Castle also has the advantage of being a covert channel that is largely implemented with off-the-shelf software components with only a few hundred lines of code dedicated to encoding and desktop automation scripting. Desktop automation tools are already commonly used by gamers; the game and game-specific mods (*e.g.,* replay decoder and map editor) are widespread enough to warrant little suspicion from censors since (*e.g.,* Aeons is installed by millions of users worldwide). Castle's small code base also makes it easy to distribute via hard to block asynchronous methods – *e.g.,* through collage [44], email, instant messaging, etc.

6.3. Avoiding covert channel pitfalls

Geddes *et al.* highlight three key mismatches between covert channels and cover traffic which make these look-like-something circumvention tools detectable to external observers [18]. Here we discuss how Castle avoids each of these three mismatches.

The architecture mismatch. Games provide agility in terms of architecture that few other channels provide. They often operate in client-server mode on publisher-hosted game servers and in peer-to-peer mode in direct-connect multi-player games. Our proxying approach can operate in whichever mode is the dominant, and in the presence of blocking can even shift (*e.g.,* from client-server mode to peer-to-peer mode).

The channel mismatch. While game data is typically communicated over a UDP channel, it is unresilient to packet loss unlike other UDP-based channels (*e.g.,* VoIP), thus clients come with the ability to handle packet losses and retransmissions. Further, they also guarantee in-order delivery and processing of sent data. This makes it useful as a covert channel for proxied TCP connections which require reliable transmission. Therefore, attacks that allow the censor to drop traffic to levels which are tolerable to legitimate players (but intolerable to Castle users) are not possible.

The content mismatch. Content mismatches arise when the content being embedded in the covert channel changes the flow-level features of the channel. Since the flow-level features of real-time strategy games are strongly dependent on many parameters (identified above), they are highly variable. We have shown that Castle, under every configuration, generates traffic that is well within this variance.

7. Extensibility of Castle

In order to evaluate the extensibility of Castle to new real-time strategy titles, we considered the time required and the development procedure used by an under-graduate researcher to complete a basic port of Castle over two extremely popular (over 8.5 million copies sold) closed-source real-time strategy games from two different development studios – Aeons and Conquerors.

Castle attempts to be easily adapted to many real-time strategy games by only utilizing the common command structure for encoding and replay files for decoding. As a consequence, it was possible to port Castle to Aeons and Conquerors in under 6 hours per game. The three main phases for porting Castle to a new real-time strategy game are map creation, configuring the automation toolkit, and decoding replay files.

Map creation. In some real-time strategy games where game maps are stored in easy to read formats (*e.g.,* 0-A.D.), maps for use with Castle can be generated via simple scripts. In others which use proprietary map storage formats (e.g., Aeons and Conquerors), the developer is required to manually place units at specific (known) locations on the game map. In such cases, to reduce the effort required for this time consuming process, Castle currently provides an easy to configure AutoHotKey script to automatically generate maps via desktop automation and the GUI map editor of any real-time strategy game.

Desktop automation. To allow Castle to execute commands within the game, desktop automation tools have to be integrated with the real-time strategy game. During this process the developer is required to supply configuration parameters including maximum number of selectable units, click sending mode (AutoHotKey provides four modes. Selection of a mode is dependent on the type of application and DirectX version), window title, and a sutiable inter-click speed (*e.g.,* Conquerors blocks clicks from Castle in its fastest configuration).

Decoding replay files. Finally, in order to decode data sent by a Castle client, Castle needs to be able to retrieve data stored in the form of in-game commands in game replay files. Fortunately, replay decoders are already available in the hacking/gaming community for many popular games (e.g., Aeons). For other games without an available decoder (e.g., Conquerors), gaming and hacking forums such as [45], [46], [47] provide techniques and support for building a decoder. Additionally, the process is made simpler since it is sufficient for the developer to be able to extract `MOVE` or `SET-RALLY-POINT` commands (rather than needing the ability to decode any command stored in a replay file). Finally, since most real-time strategy games from a studio use the same replay format, the overhead of decoding replay files is amortized over the entire set of users using real-time strategy games from the same studio as a cover.

7.1. The consequence of extensibility

Although individual game titles do not necessarily represent a high collateral damage for a censor in the event that they are blocked, Castle presents a simple framework to convert each title into an ephemeral and effective covert channel, with minimal development overhead. This, along with the fact that most newly released real-time strategy games are potential covert channels, makes Castle particularly useful in the arms race engaging censors and developers. In particular, Castle is a proof-of-concept censorship circumvention tool introducing the feature of *portability*. This portability increases potential to achieve a favorable asymmetry in the arms race – *i.e.,* it is easier and cheaper for a circumvention tool developer to create a new covert channel than it is for a censor to detect it.

8. Performance Evaluation

Without any game-specific modifications, Castle offers performance amenable to transfer of textual data (*e.g.,* tweets, e-mail, news articles)[2] and even bootstrap higher bandwidth secure communication channels (e.g., for distribution of Tor Bridge IP addresses).

8.1. Castle throughput

The throughput achieved by Castle is dependent on two certain game characteristics – maximum number of units on a game screen and maximum number of selectable units in a single command.

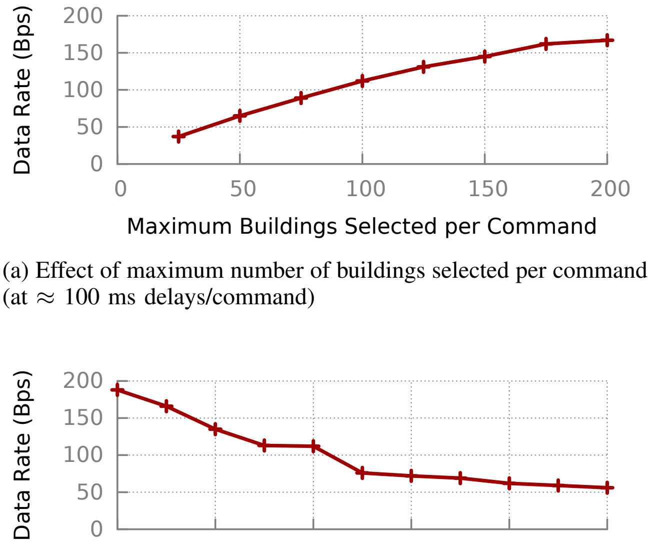
- **Maximum units per game screen (n):** Depending on the size of the units used within the game and the layout of the game screen, the number of units that may be placed within a Castle map for the game varies. For example, as illustrated in Table 2, 0-A.D. is able to fit up to 1600 units on a Castle map, while Aeons and Conquerors allow only up to 435 and 416 units, respectively.
- **Maximum selectable units per command (k):** Some games impose limitations on the number of units that may be commanded at once. For example, 0-A.D. allows only up to 200 units/command and Conquerors allows only up to 40 units/command.

Since Castle is able to send upto $\frac{\sum_{i=1}^{k} \log_2 \binom{n}{i}}{k \times 8}$ bytes per command on average, these parameters directly affect its throughput. Given the game specific parameters, we are able to achieve averages of approximately 65 bytes/command for 0-A.D., 39 bytes/command for Aeons, and 14 bytes/command for Conquerors.

[2] The success of the voices feeds [48] during the Arab Spring shows that in some situations textual data is enough to get information out.

Game	Max. Units/Screen	Max. Selectable Units/Command	Avg. Vanilla Castle Bandwidth (Bps)	Max. Vanilla Castle Bandwidth (Bps)	Avg. 10KB Article Transfer Time (sec)
0-A.D.	1600	200	190	320	52
Aeons	435	435	130	179	77
Conquerors	416	40	42	70	238

TABLE 2: Game imposed limitations and their effect on vanilla Castle performance.



(a) Effect of maximum number of buildings selected per command (at ≈ 100 ms delays/command)

(b) Effect of time delays per command (at ≤ 200 buildings selected/command)

Figure 6: Throughput of Castle implemented over 0 A.D. under various configurations

In addition, throughput is also dependent on the time required by the desktop automation tool to perform the actions required to issue a command (i.e., click each unit to be selected and click the target coordinate) and the time delay issued between successive commands.

To illustrate the effects of these parameters within a particular game, in Figure 6, we see their effect on Castle transfer rates when implemented over 0 A.D. Specifically, Figure 6a shows the effect of increasing the maximum number of buildings selected in a single command and Figure 6b demonstrates the effect of increasing the delay between commands.

At the average performance configurations for 0-A.D., Aeons, and Conquerors, vanilla Castle[3] requires 52, 77, and 238 seconds for transferring a short 10KB file, respectively. This is suitable for asynchronous communication and bootstrapping higher bandwidth communication channels.

8.2. Improving Castle throughput

There are several approaches to improve the throughput of castle.

Parallel requests: Since modern real-time strategy games allow eight or more players to participate in a single multiplayer game, it is possible for one censored user to encode content requests to as many as seven (or more) proxies in parallel – achieving at least a seven fold increase in throughput. This is particularly useful in the context of web data, where requests are easy to parallelize.

Game specific enhancements: Many real-time strategy games offer features that are not universal. For instance, many games provide *trigger* controls to map designers – i.e., a feature that allows map designers to specify responses to player actions (if a player performs action x, action y happens to unit z). Such features allow Castle to encode significantly more data than currently possible – e.g., Castle could use a hierarchical encoding structure if camera motion actions are permitted in trigger systems. Other games provide significantly more comprehensive replay information and include preserving the order of clicks performed by opponents. This allows castle to achieve significantly more Bytes per command ($O(\log_2 P(n,k))$) than it currently does ($O(\log_2 C(n,k))$). An example of such a non-universal feature used to improve Castle's throughput is demonstrated for Aeons in Section 8.3.

Content compression: Castle proxies may improve performance by compressing requested content before encoding. In the context of web data, the proxies may also pre-render and compress content before sending to the Castle receiver (*e.g.,* as was done by the Opera mobile browser [49]).

8.3. Game-specific enhancements for Castle

In this section, we show that the performance of Castle can be improved significantly through simple game-specific tweaks. To be able to observe the impact of these game-specific modifications, Aeons was used as the channel for vanilla Castle and Castle with Aeons-specific modifications. The game-specific modifications were introduced and implemented for Castle in just under three hours by an undergraduate researcher.

The low throughput of Castle over Aeons was because Aeons had larger units than 0 A.D., thereby allowing players to place only 435 units within a single screen (as opposed to 1,600 for 0 A.D.). As a result, the throughput of vanilla Castle was only ≈ 38 bytes/command (i.e., ≈ 130 bytes/second) – i.e., with the maximum command rate of AutoHotkey and selection of up to 435 units/command.

[3] We refer to the original Castle design as described in Section 3 (without any additional enhancements) as vanilla Castle.

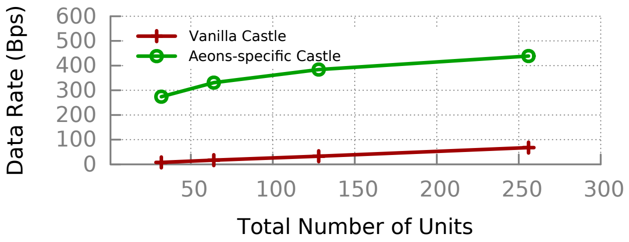


Figure 7: Throughput of Castle (with and without enhancements) implemented over closed-source Aeons

A quick investigation into the Aeons replay mode and save-game files revealed that even the selection of a single unit was communicated over the network and logged by other players. We exploit this fact by creating a set of 2^m units (256 in our case) and mapping each unit to an $m-bit$ value (*i.e.,* a byte). We then sequentially transfer the data byte-by-byte via selecting the unit corresponding to the byte to be encoded.

This encoding allowed AutoHotkey to issue commands at a significantly faster rate than before (a command was now just a single mouse click, as opposed to up to 435 key presses and clicks). At AutoHotkey's fastest mouse click rate and $m = 8$, this encoding achieves a throughput of up to 3 KByte/second. However, in order to more closely mimic the command rate and traffic generated by a skilled human player, a delay of 3 ms per command is added.

In Figure 7, we show the effect of this game-specific modification on the throughput of Castle. From the same figure, we can also observe the effect of varying the total number of units with vanilla Castle and the Aeons-specific version of Castle. We see that increasing n results in a linearly increasing throughput for vanilla Castle, and a logarithmically increasing throughput for Aeons-specific Castle. However, because the cross-over point of these functions is higher than the game allows, Aeons-specific Castle always achieves better throughput for Aeons.

8.4. Castle system requirements

From previous work on the pitfalls of circumvention tools [17], [18], it is clear that any *look-like-something* covert channel should transmit data via an off-the-shelf implementation of the cover protocol. This results in computation resource consumption since the implementation must always be executed in the background during transmission.

While video games are more resource-intensive than other cover applications, real-time strategy games are the least demanding of the many genres of video games. For example, 19 of the top 20 best-selling real-time strategy games of all-time do not require graphics cards (beyond standard integrated cards) and have fairly modest memory requirements (under 2 GB RAM). Since Castle does not add much computational overhead beyond the running of the cover video game itself, it is usable by a large number of censored users.

9. Castle and the State-of-the-art

In this section we compare the design methodology, extensibility, and performance of Castle with the state-of-the-art in *look-like-something* circumvention systems: Rook [30], FreeWave [13], and Skypemorph [11].

Design methodology and adversary model. Like Castle, both FreeWave and Rook actually use their cover protocols (VoIP and video games, respectively) – *i.e.,* they insert covert data via the application layer, rather than directly at the transport layer (as is done by Skypemorph). As a result, standard mimicry detection attacks such as IP/Port filtering and active probing are unable to distinguish the use of the covert channel from the cover channel (while they succeed against Skypemorph [17], [18]).

While both FreeWave and Castle rely on the fact that their main communication channels are encrypted, Rook does not. Rather, Rook focuses on achieving steganographic security (resulting in a much stronger adversary); even if the adversary is able to observe the unencrypted communications between a Rook server and client, it is still unable to distinguish the usage of Rook from the cover video game. This is not the case with Castle or FreeWave – *i.e.,* an adversary that is able to observe unencrypted communication between the proxy and the client is easily able to distinguish the covert channel from the cover channel.

Although Castle, Rook, and FreeWave all use UDP communication channels, FreeWave is unable to avoid detection by active attacks which perturb the network traffic by delaying or dropping packets. This is a result in a mismatch of reliability requirements between the cover protocol (multimedia VoIP) and the covert channel data which demands higher reliability. In contrast, Castle and Rook leverage video games as a cover protocol where reliability is built into the application layer (by default in Castle and by the covert protocol in Rook).

Extensibility. Since Castle, Rook, and FreeWave insert covert data via the application layer, they are extensible to varying degrees. Skypemorph on the other hand is built to mimic the Skype protocol, and is therefore not extensible.

As demonstrated in Section 7, Castle is easily extensible to any real-time strategy game which has (1) a map editor, (2) a `MOVE` or `SET-RALLY-POINT` command, and (3) a replay file decoder. In contrast, extensibility of Rook and Freewave is hindered by: requiring a deep understanding of the internal networking protocol used by the cover channel (Rook), or the absence of a large number of cover applications to extend to (FreeWave).

Performance. In terms of throughput, both VoIP based covert channels – FreeWave and Skypemorph perform better than Castle and Rook. Skypemorph is able to achieve a covert data transmission rate of 34 KBps, while FreeWave achieves 2.4 KBps. The large difference between the two systems should be attributed to the significantly stronger adversary model used by FreeWave (FreeWave is built to be secure against active probing).

The throughput achieved by Castle is dependent on the characteristics of the game being used as a cover channel. In our experiments considering three different cover video games, we observed covert data transfer rates in the range of 42 - 320 Bps without any modifications to Castle. With game specific modifications, we were able to achieve up to 435 Bps. Since Rook focuses on steganographic security (a much more powerful adversary than Castle), it also suffers from lower throughput – *i.e.,* between 3 and 5 Bps.

10. Conclusions

In this paper we have presented Castle, a general approach for creating covert channels using real-time strategy games as a cover for covert communications. We demonstrate our approach by prototyping on three different games with minimal additional development overhead and show its resilience to a network adversary.

We argue that the popularity, availability, and generic functionalities of modern games make them an effective circumvention tool in the arms-race against censors. Specifically, our results show that Castle is:

- **Portable and Extensible:** Incorporating new closed-source games as covert channels for Castle requires only a few hours of developer time – including the addition of title-specific enhancements for increased throughput.
- **Secure:** Castle is resistant to attacks such as IP/port filtering and deep-packet inspection since it actually executes the game application. More complicated and expensive attacks such as traffic analysis attacks are avoided due to the high variability of standard game flows. In addition, Castle is also resilient against active and application-layer attacks.
- **Usable:** Even without any game-specific modifications, Castle is able to provide throughput sufficient for transfer of textual data and bootstrapping higher-bandwidth channels.

The results presented in this work motivates two independent future research directions. First, Castle demonstrates that portability is possible in circumvention tools. Therefore, extending our work to different classes of applications which may enable higher throughput rates may yield a more powerful defense against censorship. Second, integrating the Castle approach into platforms to make it usable to users *e.g.,* via a Web browser plug-in or integration with the suite of Tor Pluggable Transports [50].

Code and data release: The source code of Castle (not including game specific code – *e.g.,* replay decoders, map generators, etc.) is available under the CRAPL license [4] at https://github.com/bridgar/Castle-Covert-Channel.

[4] http://matt.might.net/articles/crapl/

Acknowlegements

This material is based upon work supported by the National Science Foundation under grant numbers CNS-1350720 and CNS-1518845, a Google Faculty Research Award, and an Open Technology Fund Emerging Technology Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Google, or the Open Technology Fund.

References

[1] Philip N Howard, Aiden Duffy, Deen Freelon, Muzammil Hussain, Will Mari, and Marwa Mazaid. Opening Closed Regimes: What Was the Role of Social Media During the Arab Spring? 2011.

[2] Lev Grossman. Iran protests: Twitter, the Medium of the Movement. *Time Magazine*, 17, 2009.

[3] ONI Research Profile: Burma. http://opennet.net/research/profiles/burma, 2012.

[4] H. Noman. Dubai Free Zone No Longer Has Filter-Free Internet Access. http://opennet.net/blog/2008/04/dubai-free-zone-no-longer-has-filter-free-internet-access.

[5] ONI Research Profile: Indonesia. http://opennet.net/research/profiles/indonesia, 2012.

[6] Glenn Greenwald and Ewen MacAskill. NSA Prism Taps into User Data of Apple, Google, and Others. http://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data.

[7] A. Dainotti, C. Squarcella, E. Aben, K.C. Claffy, M. Chiesa, M. Russo, and A. Pescapé. Analysis of Country-wide Internet Outages Caused by Censorship. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 1–18. ACM, 2011.

[8] E. Zmijewski. Accidentally Importing Censorship. Renesys blog. http://www.renesys.com/blog/2010/03/fouling-the-global-nest.shtml, 2010.

[9] Rensys Blog. Pakistan Hijacks YouTube. http://www.renesys.com/blog/2008/02/pakistan_hijacks_youtube_1.shtml.

[10] Freedom House. Freedom on the Net 2014. https://freedomhouse.org/sites/default/files/FOTN_2014_Full_Report_compressedv2_0.pdf.

[11] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. SkypeMorph: Protocol Obfuscation for Tor Bridges. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 97–108, New York, NY, USA, 2012. ACM.

[12] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: A Camouflage Proxy for the Tor Anonymity System. In *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, October 2012.

[13] Amir Houmansadr, Thomas Riedl, Nikita Borisov, and Andrew Singer. I Want my Voice to be Heard: IP over Voice-over-IP for Unobservable Censorship Circumvention. In *Proceedings of the Network and Distributed System Security Symposium - NDSS'13*. Internet Society, February 2013.

[14] Tor Blocked: The Tor Blog. https://blog.torproject.org/category/tags/tor-blocked. Accessed: 2015-02-22.

[15] The Tor Project. Gfw actively probes obfs2 bridges. https://trac.torproject.org/projects/tor/ticket/8591.

[16] Great Fire. China just blocked thousands of websites. https://en.greatfire.org/blog/2014/nov/china-just-blocked-thousands-websites.

[17] A. Houmansadr, C. Brubaker, and V. Shmatikov. The Parrot Is Dead: Observing Unobservable Network Communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79, May 2013.

[18] John Geddes, Max Schuchard, and Nicholas Hopper. Cover Your ACKs: Pitfalls of Covert Channel Censorship Circumvention. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &; Communications Security*, CCS '13, pages 361–372, New York, NY, USA, 2013. ACM.

[19] Steam, The Ultimate Online Game Platform. http://store.steampowered.com/about/. Accessed: 2015-02-22.

[20] Chris Chambers, Wu-chang Feng, Wu-chi Feng, and Debanjan Saha. Mitigating information exposure to cheaters in real-time strategy games. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '05, pages 7–12, New York, NY, USA, 2005. ACM.

[21] Matt Pritchard. How to hurt the hackers: The scoop on internet cheating and how you can combat it. *Gamasutra, July*, 24, 2000.

[22] Microsft MSDN. Networking for Games. https://msdn.microsoft.com/en-us/library/windows/apps/mt210805.aspx.

[23] Steam. Steamworks API Overview. https://partner.steamgames.com/documentation/api.

[24] Marc Liberatore and Brian Neil Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 255–263, New York, NY, USA, 2006. ACM.

[25] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial na&#239;ve-bayes classifier. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 31–42, New York, NY, USA, 2009. ACM.

[26] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Computer Security ESORICS 2006*, volume 4189 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin Heidelberg, 2006.

[27] 0 A.D.: A Free, Open-Source Game of Ancient Warfare. http://play0ad.com. Accessed: 2015-02-22.

[28] S. Zander, G. Armitage, and P. Branch. Covert channels in multiplayer first person shooter online games. In *Local Computer Networks, 2008. LCN 2008. 33rd IEEE Conference on*, pages 215–222, Oct 2008.

[29] S. Zander, G. Armitage, and P. Branch. Reliable transmission over covert channels in first person shooter multiplayer games. In *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, pages 169–176, Oct 2009.

[30] Paul Vines and Tadayoshi Kohno. Rook: Using video games as a low-bandwidth censorship resistant communication platform. 2015.

[31] Cisco Inc. Cisco ios netflow. http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html.

[32] Wikipedia. List of best-selling PC games. http://en.wikipedia.org/wiki/List_of_best-selling_PC_games.

[33] Paul Bettner and Mark Terrano. 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. *Presented at GDC2001*, 2:30p, 2001.

[34] Donald E Knuth. The Art of Computer Programming, Volume 4: Generating all Combinations and Partitions, Fascicle 3, 2005.

[35] BridgeDB: The Bridge Distribution Database. https://gitweb.torproject.org/bridgedb.git/tree. Accessed: 2015-02-22.

[36] AutoHotkey. Macro and Automation Windows Scripting Language. http://www.autohotkey.com/.

[37] Amer Ajami. Singapore Bans Half-Life. http://www.gamespot.com/articles/singapore-bans-half-life/1100-2446255/.

[38] Amer Ajami. Half-Life Ban Lifted. http://www.gamespot.com/articles/half-life-ban-lifted/1100-2446436/.

[39] The New Zealand Herald. Ban Lifted on Xbox Game with Sex Scene. http://www.nzherald.co.nz/asia/news/article.cfm?l_id=3&objectid=10476791.

[40] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 143–157, San Diego, CA, August 2014. USENIX Association.

[41] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, WPES '11, pages 103–114, New York, NY, USA, 2011. ACM.

[42] Tao Wang and Ian Goldberg. Improved website fingerprinting on tor. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 201–212, New York, NY, USA, 2013. ACM.

[43] Tao Wang. Website Fingerprinting. https://cs.uwaterloo.ca/~t55wang/.

[44] Sam Burnett, Nick Feamster, and Santosh Vempala. Circumventing censorship with collage. *SIGCOMM Comput. Commun. Rev.*, 40(4):471–472, August 2010.

[45] MPGH. MultiPlayer Game Hacking and Cheats. http://www.mpgh.net.

[46] AoC Zone. AoC Zone. http://www.aoczone.net.

[47] Game Replays. Game Replays. http://www.gamereplays.org.

[48] The Voices Feeds. http://johnscottrailton.com/the-voices-feeds/. Accessed: 2015-02-22.

[49] Opera Turbo. Opera Help: Opera Turbo. http://help.opera.com/Mac/11.60/en/turbo.html.

[50] Tor Project: Pluggable Transport. https://www.torproject.org/docs/pluggable-transports.html.en. Accessed: 2015-02-22.