Minecraft Altered Skin Channel (MASC)

Sam Abrams, Brandon Keller, Kenneth Nero, Gino Placella, Daryl Johnson

▶ **To cite this version:**

Sam Abrams, Brandon Keller, Kenneth Nero, Gino Placella, Daryl Johnson. Minecraft Altered Skin Channel (MASC). 36th IFIP International Conference on ICT Systems Security and Privacy Protection (SEC), Jun 2021, Oslo, Norway. pp.134-145, 10.1007/978-3-030-78120-0_9 . hal-03746056

HAL Id: hal-03746056
https://inria.hal.science/hal-03746056v1

Submitted on 4 Aug 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Minecraft Altered Skin Channel (MASC)

Sam Abrams, Brandon Keller, Kenneth Nero, Gino Placella, and Daryl Johnson

Rochester Institute of Technology, Rochester NY 14623 USA
{sxa7263,bnk5096,kbn1798,gap6768,Daryl.Johnson}@rit.edu

Abstract. Governments across the world are increasingly pushing for encryption backdoors and other privacy breaking features. One possible method to maintain privacy in a post-encryption environment is to utilize covert channels as a means to hide traffic from monitoring authorities. Covert channels have often been used to hide unwanted or prohibited web traffic, however, the traffic need not be malicious for there to be a necessity for privacy.
In this paper we explore a novel covert channel named Minecraft Altered Skin Channel (MASC) that exploits the inherent inefficiencies within Minecraft skin images. Data is transferred by reflecting a modified skin image off of the publicly accessible Minecraft Skin servers. We utilize a custom steganographic algorithm and a public Application Programming Interface (API), which when combined, grant the ability to quickly and asynchronously send and receive encoded messages without any direct communication between the two or more participating parties. We demonstrate the effectiveness of the scheme by implementing an efficient sender-receiver proof of concept. Finally, we go on to empirically analyze the program's results and show the comparatively high bitrate of the channel.

Keywords: Covert Channel · Security · Steganography · Minecraft.

1 Introduction

Covert channels existed long before the internet. However, only since 1973 and the publication of [5] have covert channels become almost exclusively associated with digital networking. The Internet Corporation for Assigned Names and Numbers (ICANN) gives the example of a hidden pocket in a briefcase as a physical implementation of such a channel; by hiding something within another means of transport, a hidden, or covert, channel is created [6]. While the medium may have changed, the need to secretly transport data has remained. In [5], the author defined "covert channels" as methods of communication that are "not intended for information transfer at all". While this definition is by no means perfect, it does manage to encapsulate nearly all modern covert channels, including our Minecraft Altered Skin Channel, MASC.

Covert channels have historically been used for two purposes: hiding malicious communication and hiding legitimate traffic that requires secrecy. Malicious covert channel based communication is present in large amount of malware,

taking advantage of hidden traffic to avoid detection by firewalls and other network inspection devices. Legitimate traffic does the same thing, but for a good reason. An example of legitimate traffic is a whistle blower in an authoritarian state that cannot have their communication detected without consequence. While both uses of covert channels exist, it is more common to see these channels used for malicious activity simply because of the volume of cybercrime and its need to avoid detection. Given this growing concern of cybercrime and increased investment into detecting these hidden transmissions, there is an ever-growing need for new covert channels across a variety of mediums that MASC seeks to fill.

The rest of this work is organized as follows, Section 2 provides background on game-based covert channels, Section 3 provides an overview of the proposed channel, Section 4 outlines the encoding and decoding process, Section 5 outlines the performance of the proposed channel, Section 6 outlines potential countermeasures that could be employed against the channel, and Section 7 provides conclusions and recommendations for future works.

2 Background

Game based covert channels are not a novel concept. There are many examples of behavioral and storage based channels implemented within video game protocols. Our initial work was strongly influenced by the Runescape channel developed by Kim *et al.*[3], a storage channel that took advantage of item placement within a player's inventory to communicate between two users. Using item placement is an obvious choice for Minecraft, a game that revolves around building and crafting, and so is not as exciting as we wanted. Additionally, using game server reliant systems severely limits throughput. Minecraft servers are limited to a 20Hz [2] refresh rate, making it incredibly difficult to pass more than a few bytes per second. For example, if we were to represent a certain item type within the game as a alphanumeric character, we could place or remove 20 blocks per second. That would limit the potential rate to 20 characters per second, or approximately 20B/s, too slow to effectively pass any data beyond a short text message. Being limited by server tick rate is a problem faced by many game based channels. Some channels increase bitrate by taking advantage of games with refresh rates as high as 128Hz, but even then the max bandwidth is relatively low. Our channel avoids the bottleneck of refresh rate by transcending the direct game servers. Instead, we communicate with the Minecraft Session servers, which store the avatar skins displayed in game.

Others have also sought to use games as a medium for covert channels. The Castle channel, proposed in [4], encodes data into player actions within RTS games. This platform relies on other players being in the same game at the same time. A limitation the asynchronous nature of our platform allows us to avoid. In another example, the proposed platform in [7] uses player movement within first person shooter games to transmit data to other players. This falls victim

to the same limitation previously discussed as movement data will need carried from a source client, to the game server, and then to receiving client.

3 Minecraft Skin Channel

Minecraft is an open world survival sandbox game developed by Mojang and initially released in November 2011 [1]. Minecraft allows for an enormous amount of user customization, including the ability for users to create and modify the texture files that determine how the game looks. The feature we leverage for our covert channel is the ability of players to change their in-game avatar's texture file. These "skin" files are created in a specific format in order to give the avatar a different appearance in-game. The current format used by Minecraft has an 832 pixel "dead space" in the 64x64 pixel texture image that is not displayed in-game or processed in any way. By using this dead space and a custom steganography algorithm, we are able to encapsulate messages into skin files with our own pixel data, thereby introducing arbitrary data into an otherwise legitimate texture file. A sample file with this dead space highlighted is shown in *Figure 1*.

Since our channel does not rely on an avatar being in game, we discovered that we can use Mojang's official Texture Servers to distribute our files without limitations. The server is simple to navigate. One player will upload a created texture under their username, which allows any player to download the same texture by referencing the username of the creator. We use this simple process to transfer the encoded skin file from the sender, through the server, into the hands of the receiver. The only time login credentials are required is for the initial upload of a texture. That means the receiver can remain totally anonymous, and even the sender can be semi-anonymous by creating an unassociated Minecraft account. This provides the users with a simple, efficient, and effectively anonymous process for sending and receiving data.

The channel operates using several command line arguments that the sender will specify. They will provide data to be encoded that is less than or equal to 4992 characters of a limited character set consisting of 39 characters, an original 'skin' file, and a seed to be used for lookup table generation. The sender's username, password, and email address are also required in order to communicate with the Mojang Texture Servers. The message will then be encoded and transferred into the 'dead' space of the provided 'skin' file. The results are uploaded to the Mojang Texture Servers and stored under the senders specified username.

In order to receive the data, the receiver needs the sender's Minecraft username and the lookup table seed. The 'skin' file is downloaded from the texture servers with two API calls. The first API call accesses the universally unique identifier based on the sender's username, then the second API call obtains the encoded texture link and downloads it. Once downloaded, the pixels held in the 'dead' space of the 'skin' file are enumerated and decoded. Once decoded, the skin file is discarded and the message is output to the command line. The process is very linear, requiring no decisions or verification, and is shown in *figure 2*.

Fig. 1. A sample Minecraft skin file. The unused pixels are shown in grey/white checkerboard

4 Encoding & Decoding

Functionality of MASC is separated into two sections: the client and the backend encoder/decoder. Our encoder uses an unrestricted form of steganography to embed six-character chunks of data into the allowed pixels. Typically, steganography must alter as few bits as possible in an image in order to maintain that image's integrity and appearance. MASC does not have this constraint. It can modify the selected pixels without fear of modifying the image's appearance as the modified pixels are not displayed at any time. Encoding works by using the pixel's color values to store a binary string, which in turn is simply a number that represents the alphanumeric string. Obtaining that number is the key to MASC's quick and space efficient operation. With that number, a custom pixel

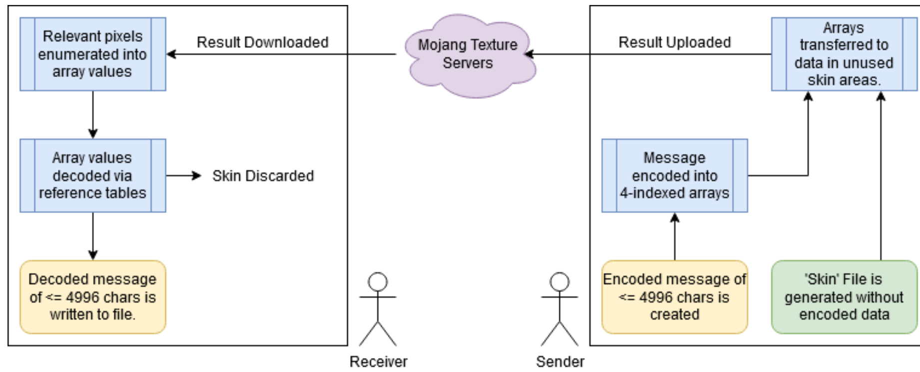

Fig. 2. A diagram exploring the communication architecture for sending and receiving messages utilizing a minecraft skin

can be created and then embedded into the base Minecraft skin image following the pixel location guidelines.

To calculate the number representing each six character string, five distinct steps must be taken. This process is shown in part in *figure 3*.

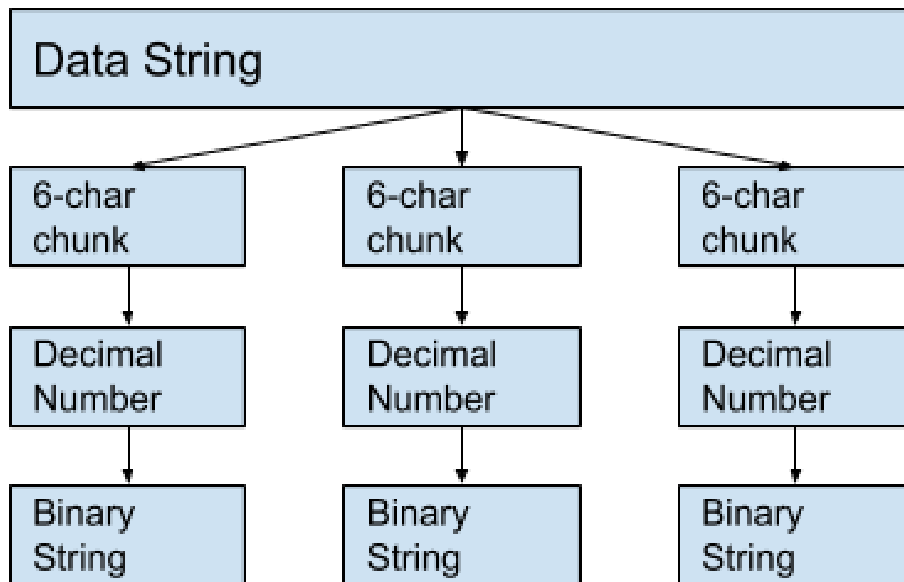

Fig. 3. A graphical representation of the pixel encoding process

4.1 Encoding

Data Processing This step is a simple validation of the data to be encoded. MASC utilizes a limited character set of 39 characters, shown in *figure 5* and *figure 6*, requiring that any unavailable characters be removed before encoding. Any of these characters found will be replaced with a space, and all lowercase letters are converted to uppercase. As there is a 4992 character limit, any data exceeding this limit is removed.

Original text: hello!my n*me is B&B
Replacement text: HELLO MY N ME IS B B

The Split Once all unrecognized characters have been removed, the data string must be split into six character chunks to be encoded. If the number of characters is not divisible by six, spaces will be prepended to the data string. This allows for even data chunks while also allowing for easy detection and removal after the decoding process.

Original text: HELLO MY NAME IS BOB
Replacement text: [␣␣␣␣HE], [LLO MY], [NAME], [IS BOB]

Calculation Each of these six-character chunks is then encoded one at a time. To begin the encoding process, each chunk is decomposed into its characters which are then operated on individually. Each of these characters is assigned a unique number from 0 to 38, determined by the lookup table, an example of which is found in *figure 5*. The lookup table is randomly generated during runtime from a seed provided by the user. Once the character's value is determined, it is multiplied by the number of permutations, including repetition, that are possible within the remaining space of the chunk. The parameters to determine this number of permutations are 39 characters in $5-i$ spaces where i is the index of the character in the six-character chunk. This multiplication by the number of permutations is essential for properly creating the unique identifier for a given six-character string. The full assignment and multiplication operation is repeated for each of the six characters, and all six values resulting from these operations are summed. This summed value becomes the numerical representation of the chunk. The general formula for an entire six-character chunk is found below.

$$\sum_{i=0}^{5} lookup(chunk[i]) * 39^{5-i}$$

Using the string "SAMUEL" as the original input, the calculation would be performed as follows. The reference table utilized for this specific example is in *figure 5*.

$$S = 6 * 90,224,199 = 541,345,194$$

where 90,224,199 is the number of permutations with n=39 & 5 items included.

$$A = 2 * 2,313,441 = 4,626,882$$

$$M = 14 * 59,319 = 830,466$$

$$U = 11 * 1,521 = 16,731$$

$$E = 0 * 39 = 0$$

$$L = 10 * 1 = 10$$

```
     541,345,194
       4,626,882
         830,466
          16,731
               0
+             10
---------------
     546,819,283
```

Following the completion of all calculations, the final result is 546,819,283 as noted above.

Binary Conversion & Color Dictionary The number representing the chunk is then converted into its binary form and padded with zeros as necessary to fit evenly into four eight bit sections. The eight bit sections are then placed into a dictionary representing a pixel's colors. The last item in Step 4 is to add one (shown as "+1") to the "A" value to ensure that the pixel is not completely transparent.

Input: 546,819,283

Binary: 00100000100101111100110011010011, where the leading zeros have been added for usability

Split: [00100000+1], [10010111], [11001100], [11010011]

Output: {A:00100001, R:10010111, G:11001100, B:11010011}

Embedding After all the data has been encoded into a list of structures representing RGB values, the list is passed to an embedding function, which simply places each pixel into the image at the next slot available from the allowed pixels list. A simplified diagram of this process is shown in *figure 4*.

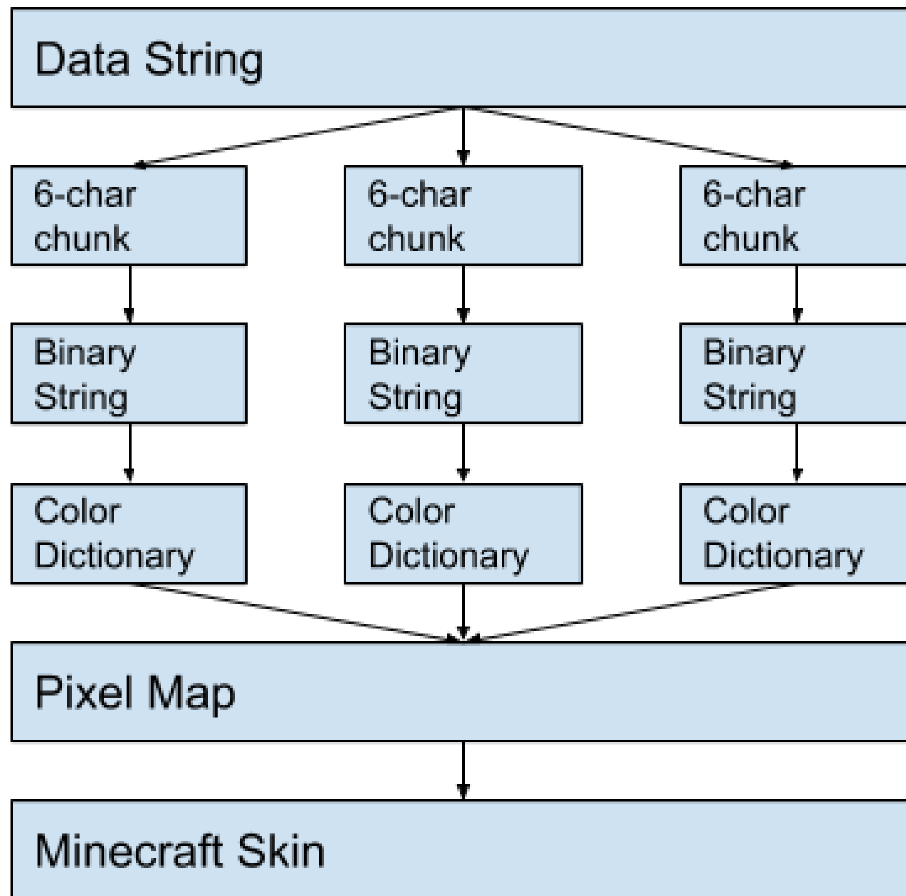

Fig. 4. A simplified representation of the encoding process. The Binary String refers to the binary representation of the permutation number.

4.2 Decoding

Decoding is a quick matter of reversing the encoding process. One section of the process is worth expanding upon, as the reversal of the forward process is slightly different than what may be intuitive. When converting from a large number back to a six character chunk, order of operations is from first character to last, rather than a true inverse of the encoding process. The reverse reference table utilized in this specific example is in *figure 6*. This table is a completely reversed table of the one generated for encoding. Every other operation is inverted as expected: multiplication becomes division and addition becomes subtraction. Comparing the example below to the example in the "Calculation" section shows how each operation is repeated in inverse.

Input: 546,819,283

$$546,819,283/90,224,199 = 6- > S$$

$$546,819,283 - (90,224,199 * 6) = 5,474,089$$

$$5,474,089/2,313,441 = 2- > A$$

$$5,474,089 - (2,313,441 * 2) = 847,207$$

$$847,207/59,319 = 14- > M$$

$$847,207 - (59,319 * 14) = 16,741$$

$$16,741/1,521 = 11- > U$$

$$16,741 - (1,521 * 11) = 10$$

$$10/39 = 0- > E$$

$$10 - (39 * 0) = 10$$

$$10/1 = 10- > L$$

Output: SAMUEL

5 Performance

What makes our covert channel unique compared to other similar game based covert channels is its performance. Encoding a maximum length message of 4,992 characters took 12 milliseconds, and decoding that same message took only 8.4 milliseconds. Combined with uploading and downloading the file, the overall transmission time for our message is approximately half a second, dependent largely upon users' internet connections.

Hahn *et al.* note that their channel, Castle, a covert channel implemented in real-time strategy games, “has about 100x more bandwidth than other proposed game-based covert channels” [4]. Castle is capable of 50-200B/s, and with tuning can transmit up to 400B/s. Our initial estimate for our channel was 166B/s, with a 30 second transmission time. Since that estimate, we have completely revised our encoding, decoding, and transmission algorithm, resulting in a current bandwidth of 9,984 B/s, approximately 60 times more efficient than previously estimated. This estimate was calculated by measuring the time for encoding data, uploading the skin, downloading the skin, and decoding the file compared to the total amount of bytes worth of text that can be conveyed. Compared to the “52 seconds for the transfer of a short 10KB text news article” [4] using the proposed Castle covert channel, our final channel is incredibly fast, with an estimated transmission time of roughly 1.5 seconds for the same 10KB of data. More drastic is the comparison to the channel proposed in [7] which demonstrated transmission rates around 7-9 bits per second.

6 Countermeasures

MASC relies on the private infrastructure provided to the public by Mojang. At any time, Mojang could implement changes that would drastically impact the functionality of MASC. The single most effective change that could block MASC in its current implementation would be to trim the excess pixels stored in the avatar skin. Instead of having a 64x64 image, the skin could be trimmed to fit into a 32x64 image with no unused pixels. Doing so would prevent MASC from modifying the pixels without constraint as is allowed by the current format. Being required to match colors and opacity would severely limit the amount of bits we could modify in an image, and thus limit the bandwidth of the channel. Another route Mojang could take is to zero-out all unused pixels in images stored on their servers, which would accomplish the same thing as trimming without the need for change to existing skin formats.

7 Conclusion & Future Works

In this paper we have demonstrated a working covert channel taking advantage of an inefficiency discovered in Minecraft skins. Our novel character encoding scheme allows for very high bandwidth without making sacrifices to performance or compromising the underlying game's functionality. Future research into MASC could include implementing an in-game API to pull Minecraft skins from nearby players, making for a more typical network traffic sequence. Additionally, there is no method for sequential transfers of data; adding an acknowledgement function could allow for transferring more than the current limit of a single message. A final future option would be to add functionality for modifying the displayed pixels on a Minecraft skin. Doing so would be much less efficient due to needing to match the existing colors, but would allow for more data to be passed in one upload and for the noted counter measures to be rendered irrelevant.

Acknowledgments

The authors would like to thank Ryan Cervantes for his guidance and continued support throughout the research process.

References

1. Minecraft Wiki, https://minecraft.gamepedia.com/
2. Tick, https://minecraft.gamepedia.com/Tick
3. Dane, J., Kim, N.: Runescape Based Covert Channel
4. Hahn, B., Nithyanand, R., Gill, P., Johnson, R.: Games Without Frontiers: Investigating Video Games as a Covert Channel. arXiv:1503.05904 [cs] (May 2015), http://arxiv.org/abs/1503.05904, arXiv: 1503.05904

5. Lampson, B.W.: A note on the confinement problem. Communications of the ACM **16**(10), 613–615 (Oct 1973). https://doi.org/10.1145/362375.362389, https://dl.acm.org/doi/10.1145/362375.362389
6. Piscitello, D.: What Is an Internet Covert Channel? ICANN Blog (Aug 2016), https://www.icann.org/news/blog/what-is-an-internet-covert-channel
7. Zander, S., Armitage, G., Branch, P.: Covert channels in multiplayer first person shooter online games. In: 2008 33rd IEEE Conference on Local Computer Networks (LCN). pp. 215–222. IEEE, Montreal, QB, Canada (Oct 2008). https://doi.org/10.1109/LCN.2008.4664172, http://ieeexplore.ieee.org/document/4664172/

Appendix

Fig. 5. The lookup table used for encoding in the provided example

‘E’	0	‘T’	1	‘A’	2
‘O’	3	‘I’	4	‘N’	5
‘S’	6	‘R’	7	‘H’	8
‘D’	9	‘L’	10	‘U’	11
‘C’	12	‘ ’	13	‘M’	14
‘F’	15	‘Y’	16	‘W’	17
‘G’	18	‘P’	19	‘B’	20
‘V’	21	‘K’	22	‘X’	23
‘Q’	24	‘J’	25	‘Z’	26
‘0’	27	‘1’	28	‘2’	29
‘3’	30	‘4’	31	‘5’	32
‘6’	33	‘7’	34	‘8’	35
‘9’	36	‘.’	37	‘,’	38

Fig. 6. The reverse lookup table used for decoding in the provided example

0	‘E’	1	‘T’	2	‘A’
3	‘O’	4	‘I’	5	‘N’
6	‘S’	7	‘R’	8	‘H’
9	‘D’	10	‘L’	11	‘U’
12	‘C’	13	‘ ’	14	‘M’
15	‘F’	16	‘Y’	17	‘W’
18	‘G’	19	‘P’	20	‘B’
21	‘V’	22	‘K’	23	‘X’
24	‘Q’	25	‘J’	26	‘Z’
27	‘0’	28	‘1’	29	‘2’
30	‘3’	31	‘4’	32	‘5’
33	‘6’	34	‘7’	35	‘8’
36	‘9’	37	‘.’	38	‘,’