The Telltale Tool: Debunked

Edition 0.1

By Lucas Saragosa [MSV6]

Preface

This informal paper will outline principles and practice for programmers which wish to make their own tools to mod games made in the *Telltale Tool*[1]. This paper can also be useful for anyone who is interested in reverse engineering and game engines. I will assume that the reader is familiar with programming in some basic ways, but no prior knowledge is needed and I will try to keep code examples as generic and easy to read as possible. Code examples in this paper will be done in basic Java/C++ syntax. In this paper to further help your understanding, we will be developing simple tools which will work on Telltale files and formats. This will be alongside the information presented. You should ideally also have some idea about how game engines work and what they do, but this is not required and will be gone over. Understanding how rendering works, and knowing the basic things which go into rendering a game is also very helpful (such as meshes, textures, game loops, etc), but again is not required.

By the end of this paper you should be able to have a good understanding of how the game engine works and how you can use and create offline tools to mod these games to your liking.

This paper is going to try and cover modding details for all games made in this game engine, starting from the oldest prototype games such as *Telltale's Texas Hold'em*[2]. We will cover all of the games up the last one released in the engine, *Sam and Max: Remastered*.

In this paper, I will cover the following:

- Game Archive formats (.TTARCH2, .TTARCH coming soon),
- Sound bank format (FMOD format and bank format),
- Data Container formats (used in shaders and archives),
- Modified encryption,
- Script format,
- Shader pack format, meta stream format will be in the next edition.

This paper is a continuous work in progress, so there may be things which are wrong or need to be altered. Please contact me on discord *(lucassaragosa#0152)* or on instagram (@lukassaragosa).

About me

I've worked on telltale modding and have reverse engineered their game engine since 2018. I am the creator *The Telltale Inspector*, which is a modding tool which uses what is presented in this paper to let people open and modify lots of files by Telltale. I was also the lead director of the *Minecraft Story Mode Rebuilt* project, which used my tools to convert meshes from the games and import them into *Minecraft* to make a perfect recreation of the Telltale games series *Minecraft Story Mode*. This project has been a success and has raised some money in donations. More information on that project can be found here and on the discord here.

[1] : The *Telltale Tool* is the game engine which was developed by ex-game developers Telltale Games Inc. Newer companies which own the rights of games made with such tool (such as 'LCG Entertainment') may continue to make games under the Telltale name however these games are not applicable for modding with the things presented out in this paper.

[2] : This was the first game made in the engine, and was made to test out the game engines functionality before actually using it to start making other games such as Telltale's signature *Sam and Max*.

1: Terminology

Throughout this paper there is common terminology which I will frequently use and it is good that you have an idea of what I'm talking about. I will also underline some of the terminology which may not be directly related to telltale specifically but is good to know so you don't get confused.

A lot of the time when talking about parts of the game engine I will refer to actual code symbols used in the game engine which can be found and referenced using *IDA64* using the Walking Dead: Definitive Series program debug database (PDB file) which is only available to some people. This program lets you 'decompile' the game engine application and generate low level C code for the engine and all of its functions. This PDB for the game app was accidentally released with the PC version of the Walking Dead Definitive Edition and since then has been used to help create all these modding tools and this paper.

You can skip parts of this section which you already are familiar with, but I will give a basic overview which generally requires no prior knowledge. It is best you read over it even if you know about what I'm talking about.

You *should* read below for a full in depth understanding.

Computer Architecture

You should have some idea of a how a computer works. The heart of a computer, the CPU, processes instructions at an extremely fast rate (known as the clock speed) and tends to be in the range of billions per second. Almost all computers (non embedded) also consist of a GPU, which is a specific type of CPU which focuses on graphics processing (although not *just* graphics is possible on these). These tends to be a lot more powerful and are specifically designed to execute graphics code (called *shaders*) extremely well and in parallel. A very much important optimisation (meaning something which improves the speed) of CPUs and GPUs (processing units) is SIMD. These are special types of instructions which are designed to act like other instructions but act on multiple (typically 4) pieces of data at once. This can drastically improve performance especially for graphics as it can work on data types such as vectors (with X, Y, Z and W components), and

perform operations on them in just one instruction (and therefore in one *clock cycle* - which just means it happens at the clock speed of the cpu and only takes the time of one tick in the CPU/GPU). Telltale use these instructions for CPU side mathematics for preparing camera matrices and other mathematical constructs.

There is also *RAM* which stands for random access memory. This is what lets a computer have a 'memory' and store data. This type of memory can be accessed at any time while the computer is powered on. It is random access because any part of it can be accessed at any time. It can be written and read to. There tends to be a latency (*delay*) when the CPU accesses memory from here as it has to travel across longer wires and along the computers *motherboard* (the main PCB - circuit board) as opposed to right inside the CPU. Another optimisation which the CPU can do here is used what is called a *cache*. Cache memory is memory which is inside the CPU and tends to be in the range on only a few kilobytes. This memory is *extremely* fast and is a lot faster than RAM access. Computer programs (such as all game engines) utilise this (this is done by the *compiler*) to make it run faster. Data which is being processed or accessed often - for example the camera matrix in the renderer - can be stored here and will drastically improve performance. We tend to want to avoid a *cache miss* where the data we want is not in the CPU cache but is in RAM so the CPU needs to issue read from main memory (RAM).

Base 2 & Number Systems

Computers work in terms of *bits*. A bit is a zero or a one. Numbers can be represented on computers by chaining bits together. This is possible by using a different number system *base*. Day to day we use base 10. This means there are 10 digits (0 through to 9) and after the maximum digit (9) we add another column of non 0 value, so the next number up (1), and set the smallest column to 0. So after 9, we get 10. This is the the exact same for all other number systems. Computers tend to be useful when working with number systems with bases which are powers of 2, so base 2, 4, 8, 16, 32, 64, etc. Base 2 means that there are only 2 digits, 0 and 1. This lends itself easily to computers as 0 can be represented in circuits as 'off' (or low voltage) and 1 as 'on' (of slightly higher voltage). Just like base 10 and any other base, any number can be represented in base 2. Counting in base 2 is just like base 10. The first 4 numbers are 0, 1, 10, 11. As you can see, when we get to adding 1 to 1, we don't get 2 like in base 10 or any base greater than 2. At this point we have reached the highest digit (1), so we add another column and set it to the first non zero digit of the number system, which in this case is the only non zero digit, 1. And we then set the smallest column back to 0. Adding 1 to 2 (in base 2, adding 1 to 10) is easy as we have a 0 in the smallest column so that just increments to 1 resulting in 11.

The *bit width* of a number in computers is important. There are 4 main types of simple binary numbers in computers (in programming these are *types* - types of data), the byte, the short, the int and the long (or long integer). A byte is a collection of 8 bits. Similarly, a short has 16 bits, an integer (int) 32 and a long 64 bits. There are of course types with more numbers of bits but these are slower for computers because they are so big. But 64 bits is more than enough, a 64 bit number can represent any base 10 number from -9223372036854775808 to 9223372036854775808. Very rarely will you encounter numbers which are larger than this. Think of bit width as the maximum number of digits a specific number can have. The bit width is this maximum number of digits in a base 2 number of that integer type.

Something very important is that when we count in computing we always start with 0, not 1. Otherwise we are wasting a value. Keep this in mind and engrave it, if you are new to working with computers.

The term 64 bit is frequently used in computing and you may have heard of the term *64 bit computers* or *32 bit computers*. This just refers the the number of bits that an address in RAM has. 32 bit computers have 32 bit addresses in RAM (which only accounts for around 4GB of RAM maximum, as the maximum number in base 2 in 32 bits is 4GB (for unsigned numbers)). Same for 64 bit. 64 bit computers can have thousands of terabytes of RAM, so in the future there is likely *never* going to be 128bit or higher addresses - at least in normal everyday devices.

An address in RAM is just a locator for a single *byte*. Kilobytes and megabytes are just terms of large numbers of bytes. 1 kilobyte is 1024 bytes (a lot of the time its simplified to 1000 bytes, but always under the hood it is 1024 - which is a multiple of 2). 1 megabytes is 1024 times 1024 bytes, or 1024 kilobytes. Gigabytes are 1024 megabytes and so on.

Base 8 and base 16 are also frequently used. Base 8 has digits 0 through 7 and base 16 has digits 0 through 9 and then A to F - as there are not 16 digit symbols so we use letters for the remaining 6. Base 2 is known as *binary*, base 8 is known as *octal* and base 16 is known as *hexadecimal* or *hex*. Base 16 will be used in this paper a lot. The number 16 in base 16 will be 10. Any number where it is equal to the base will be 10 if it uses 0 and 1 as the first two digit symbols. However the number 15 is *F*, F is the number 15 in base 10. This will take time to get your head around, but becomes very intuitive. A lot of the numbers you will see in this paper are in hex, because it lends itself very well to computers. 16 is a power of 2, so one byte can be represented by just 2 hex digits. This is because one byte is 8 bits, and one hex digit can have values 0 to 15 (F) therefore in binary 15 is 1111, which is 4 bits, so we can fit two of these in one byte. Therefore representing for example a

long (64 bits) is just 8 hex digits which is a lot better than around 19 in base 10. Base 8 is known as octal and won't be used in this paper, and is rarely used.

In representing different bases it can get confusing knowing for example what value '10' represents as if it was base 16 it would represent 16, base 8 8, and base 2, well, 2. To fix this, binary numbers are prefixed with '0b', hex numbers prefixed with '0x' and octal numbers prefixed with just a 0. For example, *0x20* is the number 32, as there are 2 of the 16 columns ('tens column'). *0b100* is the number 4 in base 10. *010* is the number 8 in base 8 (octal).

Endianness

Endianness is also a very important part of computer systems. Machines can be either *little* or *big* endian. This term refers to the order in which bytes are stored in computers. Computers work with data all in terms of bytes, which are just 8 bits stringed (connected) together. This is the smallest unit of data there is. In any integer or number which is greater than 1 byte there will be an associated ordering, for example the binary number 1111 1111 1111 1010 requires 2 bytes - each of 8 bits as it is 16 bits long. How do we know if the 1111 1111 is stored in memory first, or if the 1111 1010 is stored first?

In *little* endian the byte with the least significant bit (LSB) - the 'units' column, the smallest digit, comes **first**. Therefore we can infer that the byte with the most significant bit (MSB) - the Nth column where N is the bit width - is the last byte in memory. When we say 'in memory' we just mean in the address space, the first byte will have the lowest memory address (eg address 102) and the last byte will have the highest memory address (eg address 103, if the number type is a *short* - comprising of 2 bytes). Big endian the exact same but reversed, and from a hex editor looks 'ordered': the most significant bit's byte comes first.

Most machines use little endian, probably because the lowest bit tends to hold more significance across all data types. For example the boolean (true or false) uses the LSB to determine its value (tends to, mainly booleans are 0 for false and anything else for true). With 1 being true and 0 being false.

Your device is probably little endian. Playstation 3 and Xbox 360 are big endian as well as all PowerPC devices (eg Motorola).

Telltale's engine have everything pretty much in only little endian. Big endian will be used on PS3 games for some types of file.

Signed vs Unsigned

There tend to be two types of each bit width type of number, signed and unsigned. So there is a signed integer (4 bytes) and also an unsigned integer (4 bytes). Signed integers are still 4 bytes - 32 bits - but can store both positive and negative numbers. Unsigned numbers can only store positive numbers and use all the bits for that number. Signed numbers can be either *one's complement* or *two's complement*. One's complement just uses the MSB as the sign value: 1 being negative and 0 positive. However because of this you waste an extra value: you can represent 0 and -0 which are the same number. Two's complement fixes this by a simple trick which won't be covered here and is out of the scope but can be explained by papers and Wikipedia.

Floating Point

Numbers which comprise of a decimal point (or a floating point) require a specific type known as a *float* which is 32 bit longs. There is another version, the *double* which is 64 bits and therefore more accurate. This is quite complex and uses exponents, but all you need to know is it allows for numbers such as 0.00001 to be represented in binary systems. Floats are *always* signed.

Game Engines

Game engines can be thought of as data processing pipelines: they have inputs - game data - and they process that along with user input to create outputs such as render frames which are eventually presented to the user (sound can be seen as an output too). A game engine consists of a *game loop,* which is a programming construct in which we keep running the same code (more or less) each *frame* and this keeps happening as the game runs. A *frame* is an abstract term but can simply be thought of as a simply image along with a sound snippet. The goal of a game engine and its game loop is to produce these around 30 to 60 times a second. At this frequency, the images appear to be moving and we have created the illusion of a camera.

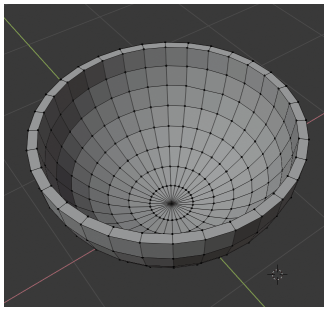
The Telltale Tool does exactly this. It has a clearly defined game loop and keeps updating all its *subsystems* each frame and then this frame is presented to the user via the graphics device and platform *window*. A *subsystem* is just a collection of

code which can be abstractly isolated. For example, the resource system is a subsystem which manages the lifetime of assets in the game engine. The graphics device is a physical graphics rendering library such as *OpenGL* or *DirectX*. A game engine should support multiple graphics libraries for different platforms, and Telltale does that. This will mean that there will be some differences between how the game engine works depending on the platform. In this paper this will come up now and then but generally the formats are platform independent.

Almost all game engines have the concept of a *scene*. A scene is just a collection of objects which may or may not be physically visible. Game engines tend have one active scene at once and the active scene can switch (eg when the story progresses in a story based game). Scene objects could be, for example, characters, UI elements, skyboxes or static meshes. These are all intrinsic parts to a game engine. Each frame, these scene objects are updated. In each frame update, there is also a concept of a *delta time*. This is just a tiny fraction of time which has passed since the last frame was executed. This difference in time can be used to calculate the change in position of a scene object, for example, of an object in the scene travelling at a constant speed. This change in time (*delta* meaning small change) is very important, without it the scene would be static and have no concept of time. This delta time will tend to be constant at the reciprocal of the frame rate, being the time period of the game engine game loop.

Rendering and Renderers

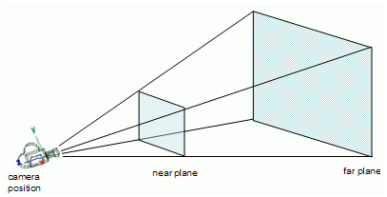
Rendering is a very important subsystem of game engines, and tends to be the most complex and most CPU/GPU consuming. A renderers main job is to take in the data about a *scene* and along with complex lighting functions create an image which is displayed to the user that frame. A *mesh* is the fundamental rendering type (meaning 'thing' in code) which describes the physical shape and textures of an object in the scene. For example, a teacup to be shown in a scene will need a mesh. This mesh will contain many *vertices*, which are 3-dimensional vectors (points) in the *object space* which are connected to form the object. *Object space* is a *cartesian* (normally) coordinate system in which the centre of the system is usually the centre of the object. All vertices in the object are in this space and can be *transformed* using matrices to get to the right point in the scene to then finally be rasterised onto the 2D plane of you computer screen/window. Below is an example,



from Blender. The black lines connect vertices to each other (these won't be seen when actually rendered in game). The points where the black lines overlap are the *vertices.* As you can see the more vertices you add the more spherical (in this case) the object looks, creating the illusion that it is smooth and therefore more realistic. Another part of meshes which is important are *faces.* A face is a collection of 3 vertices which are connected as a triangle. The faces are then coloured in using textures (images). In this case, there are no textures mapped onto the mesh yet, so the faces are coloured in grey. Another part of meshes are *UVs*, which are texture coordinates of images which then get *UV mapped* onto the mesh to create the illusion that the mesh has colours. This process can be thought of as being done each frame and this is always done on the GPU.

The second most important rendering type is the *texture*. A texture is a very broad term and there are *lots* of types of textures in rendering. The most intrinsic one is the diffuse map texture (the term map just means the texture can be thought of as a map of coordinates to colours - which essentially what a texture is). A diffuse map is any PNG or JPG image that you take with your camera on your phone and is just a collection of RGB(A) (red-green-blue-alpha) pixels. These 3 colours and alpha can describe virtually any colour seen on a computer. Each colour is known as a *channel.* The alpha channel describes how *transparent* the part of that pixel is. This is useful for when meshes are overlayed and when the renderer *blends* multiple meshes. There is so much more to rendering such as post rendering effects such as blending and antialiasing (smoothing out jagged edges on meshes using advanced algorithms), motion blur (time dependent) etc.

The last intrinsic part of a renderer is the *camera*. The camera just defines the frustum of which the user can view:



Frustum is just a shape, which is defined by two rectangles of different side and each of their corresponding corners are connected. As you can see the camera is defined by two planes (can be thought of as rectangles here): near and far. All objects and everything you see in the scene *must* lie within this shape to be seen in the viewport. Any vertices in any mesh in the current scene must lie within this shape to be viewable. If not, it is *clipped*, meaning it is not rendered in the current frame. The camera is also defined by whats known as the *FOV* ('field of view'). This is just an angle. It is the angle which defines how wide the view frustum is. This angle is the angle on the base of the frustum above between the lines going through the bottom corners of the two planes from the camera position. If this is wide, you will able to see lots. A value of about 45 degrees tends to be ok. Anything above 90 degrees starts to look a bit warped and obviously values close to zero are stupid. The camera is comprised of some other factors too, but essentially it boils down to a simple mathematical *matrix*. A matrix is just a set of numbers which transform vectors between different coordinate spaces. I won't go into detail about this, but there are many books and papers which go into detail about these. This camera matrix, along with *perspective divide* (objects further away are smaller - diving by the distance from the camera can help give an idea of this), allows you to map the 3D world in the frustum onto the 2D plane of the screen. This actually boils down to simple trigonometry and is explained again by countless books and papers.

For now you should have a basic understanding of how rendering works. Each *frame* all of the meshes in the scene go through this process of the camera mapping the vertices of the meshes to 2D points on the screen with associated colours. Rendering allows you to also execute code (this type of code is called a *shader*) at each pixel on the screen for meshes as well as for each vertex in meshes allowing for complex algorithms which can change the shape of the object at runtime to be present. For example a sphere could have a vertex shader deform is vertices to make it look jagged, all without actually changing the mesh vertices between frames. There is so much that renderers can do and in this paper you will come across lots of things which the Telltale Tool can support in relation to these rendering algorithms.

Terminology used in this paper

Below are some of the terminology which will come up a lot throughout this paper:

Hashing and the CRC64 Algorithm

Hashing is an integral part of computing and Telltale use it *massively* throughout the game engine, especially in all newer games. Hashing is the process of generating a unique number which is of fixed bit width for a set of data. For example, the *string* (characters of text stringed together) 'Hello' will have a unique number associated, which is called its hash. The string 'Hell0' with a 0 will have a different hash. This is very useful as you can use these hashes to represent file names instead of strings which for computers is very good as it can improve performance because computers like numbers a lot more than strings.

CRC64 is the hashing algorithm which telltale uses almost everywhere. It produces hashes of bit width 64, so there are many hash values (2 to the power of 64!) which any set of data can get out. Of course there can be hash *collisions* where the hash of two different sets of data are the same. This is however *extremely* rare, 1 in 'Nine quintillion two hundred twenty-three quadrillion three hundred seventy-two trillion thirty-six billion eight hundred fifty-four million seven hundred seventy-five thousand eight hundred eight' probability! So we can technically ignore this. CRC stands for *cyclic redundancy check* and is used a lot in networks to ensure that packets send between devices over the internet are not corrupted. If the hash stored in the packed is not equal to the calculated hash on the receiving end, we know the data is not the same and that we need to have it resent.

The actual algorithm doesn't need to be known and is not of importance here. The actual C++ algorithm for CRC64 is in *Appendix A*. It is important to note that telltale's implementation in their game engine uses a polynomial value ('seed') aligning with the ECMA-182 standard. So ensure in your modding tools that you implement the CRC64 algorithm with that polynomial value! (64 bit value in hex is $0x42F0E1EBA9EA3693$).

It is *very* important to note that whenever the Telltale Tool hashes a string of text, the hash is always calculated on the **lower case** version of the string. This was done presumably to reduce casing errors. This in the past has caused a lot of confusion, so in your implementation it is a good idea to create another function (eg *CRC64_CaseInsensitive*) which does this.

Header Magic Numbers

A 'magic number' or file header is just an integer (usually) at the front of every file. It allows the program which interprets the file to know if this file is valid and the user hasn't just changed the file extension.

All magic numbers in Telltale (in newer games) have their magic numbers as a 4 digit ASCII string (apart from shaders and some other formats.) This string is always *backwards* in little endian machines from a hex editor. So if you reverse this string then you will have a string of meaning which you can either check in your file interpreter against an integer or a 4 digit ASCII string.

Tables Format

When presenting file formats throughout I will present using tables using the following formats:

ID	Format Name [Format ID]	Type	Sign (S/U)
XX	Variable Name	1,2,4,8,4F,8F,Var[(/)XX],S,B,F[XX]	S/U

ID is the ID for that variable row. This ID can be referenced in the following format: *[Format ID/ID]* where the Format ID is the ID in the square brackets in the format name title row. The IDs are simply 2 digit padded numbers. The byte width is the number of bytes in that variable, which determines its type. The sign column tells you its sign, S being signed and U being unsigned. The meaning for the types in the table below:

Type	Meaning
1	Single byte (8 bits)
2	Short (16 bits)
4	Integer (32 bits)
4F	Float (32 bits)
8	Long (64 bit integer)
8F	Double (64 bit float)
Var[XX]	Raw byte data of length XX. XX as 'EOF' meaning length of rest of the file (EOF: End Of File)
S[XX]	ASCII String of length XX
B	Boolean (1 byte): ASCII of '1' as true, ASCII of '0' as false (0x31 and 0x30)
F[XX][YY]	Format, XX is the Format ID. YY is optional, default 1, and is the number of those consecutively
M[XX](S)	Magic Number String (Reversed). XX defaults to 4 if not present. (S) is what it should be.

XX above can mean two things, if it prefixed by '/' then it is the ID of the variable in the current format specifics is its length. If the variable of its length belongs to a sibling format (a format associated) then before the brackets ID the format ID is specified like in the format described above. If there is no slash then it is just the actual length in base 10 (fixed length).

Although the exception is that XX in the format type ('F') is the format ID of the format which is present at that part of the file.

Examples:

ExFormat/ID	Example Format	Type	Sign
00	First Variable	4	U
01	Second Variable	S[Example Format/00]	N/A

02	Third Variable	Var[16]	N/A
03	Fourth Variable	4F[0x2]	S
04	Fifth Variable	4[/00 + 1]	U

‘ExFormat/ID’ is the title of the ID column, ExFormat is the format ID of this new format table.

This example format has a single integer first (all in little endian) followed by a string of length of that first integer. Then there is a fixed sized buffer of 16 bytes. Then there are *2* floating points (always signed). Square brackets can follow any type, and they just mean its an array so you should read/write that many of them. Whats in the square brackets can be a straight number or it can be a reference to another format’s variable or a variable in the current format. Notice that the number in the brackets in the fourth variable is written in *hex*. Also note that in any square brackets there may be adding / subtraction etc from other IDs (referenced by /ID). Ellipsis (...) can be present in square brackets meaning a calculation is required to calculate the size (which will be described in the explanation). The last variable 04 is an array of unsigned integers which is of size first variable value plus one.

Game Engine Versions

Telltale kept updating the engine a lot. Each game differs in terms of file formats due to this. The changes tend to be small, with some new types being added and some new data being added (or removed) from file formats each version.

However there is a big major difference between *Batman Season 1 [57]* and *The Walking Dead: Michonne [56]*, being that the engine was upgraded from 32 bit to 64 bit. Along with this, the Windows platform side render API upgraded from DX9 to DX11 (which then never changed again). The mesh format changed drastically (as expected) because of this, due the rendering engine being updated.

Game References

Each game made made in the Telltale tool was made with a different version of the engine. This is because the engine was continually improving and more features were being added. Throughout this paper I will refer to the games which the section is talking about by their name and ID. The IDs are associated in the following table. The IDs are assigned in such way that a game with a larger ID value was released at a later date. Note that some games have different IDs for different episodes, meaning that between episode releases the game executable was updated.

For example if something is only relevant in games X to Y (’to’ is inclusive - both X and Y are included in this range), then I will refer in the following format (*X* to *Y* [X_{id}/Y_{id}]). Where X and Y are the games and X_{id} and Y_{id} are the associated IDs from the table below. The ID of Y will *always* be greater than X.

The Text Game ID column is used in tools made by me to reference the game instead of integer ID, this may be of use.

Appendix B lists all relevant encryption keys for each game. These keys also have an associated flag if they use the modified encryption algorithm. All of which are described in this paper.

NOTE: All of the following games tend to refer *only* to the PC (Windows) version of the game. Some PS4 versions and other platforms may differ in formats and at the moment are not included in this paper as I have not had access to them all. If a platform is specific to the game version range of a part of this paper then I will include the platform name in square brackets in the reference.

Game ID	Game Name	Text Game ID	Release Date
00	Telltale Texas Hold’em	TEXASHOLDEM	February 2005
01	Bone: Out from Boneville	BONEVILLE	September 2005
02	CSI: 3 Dimensions of Murder	CSI3DIMENSIONS	March 2006
03	Bone: The Great Cow Race	COWRACE	April 2006
04	Sam & Max: Save the World - Episode 101	SAMMAX101	October 2006
05	Sam & Max: Save the World - Episode 102	SAMMAX102	December 2006
06	Sam & Max: Save the World - Episode 103	SAMMAX103	January 2007
07	Sam & Max: Save the World - Episode 104	SAMMAX104	February 2007
08	Sam & Max: Save the World - Episode 105	SAMMAX105	March 2007
09	Sam & Max: Save the World - Episode 106	SAMMAX106	April 2007

10	CSI: Hard Evidence	CSIHARD	September 2007
11	Sam & Max: Beyond Time and Space- Episode 201	SAMMAX201	November 2007
12	Sam & Max: Beyond Time and Space- Episode 202	SAMMAX202	January 2008
13	Sam & Max: Beyond Time and Space- Episode 203	SAMMAX203	February 2008
14	Sam & Max: Beyond Time and Space- Episode 204	SAMMAX204	March 2008
15	Sam & Max: Beyond Time and Space- Episode 205	SAMMAX205	April 2008
16	Strong Bad's Cool Game for Attractive People: Episode 101	SBCG4AP101	August 2008
17	Strong Bad's Cool Game for Attractive People: Episode 102	SBCG4AP102	September 2008
18	Strong Bad's Cool Game for Attractive People: Episode 103	SBCG4AP103	October 2008
19	Strong Bad's Cool Game for Attractive People: Episode 104	SBCG4AP104	November 2008
20	Strong Bad's Cool Game for Attractive People: Episode 105	SBCG4AP105	December 2008
21	Wallace and Gromit's Grand Adventures: Episode 101	WAG101	March 2009
22	Wallace and Gromit's Grand Adventures: Episode 102	WAG102	May 2009
23	Wallace and Gromit's Grand Adventures: Episode 103	WAG103	June 2009
24	Wallace and Gromit's Grand Adventures: Episode 104	WAG104	July 2009
25	Tales of Monkey Island: Episode 101	MONKEYISLAND101	July 2009
26	Tales of Monkey Island: Episode 102	MONKEYISLAND102	August 2009
27	Tales of Monkey Island: Episode 103	MONKEYISLAND103	September 2009
28	Tales of Monkey Island: Episode 104	MONKEYISLAND104	October 2009
29	Tales of Monkey Island: Episode 105	MONKEYISLAND105	December 2009
30	CSI: Deadly Intent	CSIDEADLY	October 2009
31	Sam & Max: The Devil's Playhouse: Episode 301	SAMMAX301	April 2010
32	Sam & Max: The Devil's Playhouse: Episode 302	SAMMAX302	May 2010
33	Sam & Max: The Devil's Playhouse: Episode 303	SAMMAX303	June 2010
34	Sam & Max: The Devil's Playhouse: Episode 304	SAMMAX304	July 2010
35	Sam & Max: The Devil's Playhouse: Episode 305	SAMMAX305	August 2010
36	Hector: Badge of Carnage: Episode 101	HECTOR101	June 2010
37	Hector: Badge of Carnage: Episode 102	HECTOR102	August 2011
38	Hector: Badge of Carnage: Episode 103	HECTOR103	September 2011
39	Nelson Tethers: Puzzle Agent	GRICKLE101	June 2010

40	CSI: Fatal Conspiracy	CSIFATAL	October 2010
41	Poker Night: At the Inventory	CELEBRITYPOKER	November 2010
42	Back to the Future - The Game: Episode 101	BTTF101	December 2010
43	Back to the Future - The Game: Episode 102	BTTF102	February 2011
44	Back to the Future - The Game: Episode 103	BTTF103	March 2011
45	Back to the Future - The Game: Episode 104	BTTF104	April 2011
46	Back to the Future - The Game: Episode 105	BTTF105	June 2011
47	Puzzle Agent 2	GRICKLE102	June 2011
48	Jurassic Park: The Game	JURASSICPARK	November 2011
49	Law & Order: Legacies	LAWANDORDER	December 2011
50	The Walking Dead (Season 1)	TWD1	April 2012
51	Poker Night 2	CELEBRITYPOKER2	April 2013
52	The Wolf Among Us	FABLES	October 2013
53	The Walking Dead: Season 2	WD2	December 2013
54	Tales from the Borderlands	BORDERLANDS	November 2014
55	Game of Thrones	THRONES	December 2014
56	Minecraft: Story Mode	MCSM	October 2015
57	The Walking Dead: Michonne	MICHONNE	February 2016
58	Batman: The Telltale Series	BATMAN	August 2016
59	The Walking Dead: A New Frontier	WD3	December 2016
60	Guardians of the Galaxy: The Telltale Series	MARVEL	April 2017
61	Minecraft Story Mode: Season 2	MC2	July 2017
62	Batman: The Enemy Within	BATMAN2	August 2017
63	The Walking Dead: The Final Season	WD4	September 2018
64	The Walking Dead: Definitive Series	WDC	September 2019
65	Sam & Max: Remastered	SAMMAXREMASTERED	December 2020

2: Data Container Format

In this section I will go over Telltale's data container format. This format is used in games newer than and including *The Wolf Among Us [52/*]*. It is the file format which encloses all telltale archives, shaders, and also used in save data files. This format is responsible for the compression and encryption of data. There are 3 main modes for this format:

- Uncompressed,
- Compressed,
- Compressed & Encrypted.

If the mode is uncompressed, then the container just adds a small header to the file followed by the payload data. If not, then the file format uses a chunk approach: the data is segregated into distinct, equal size chunks of data. Each of these data chunks is then compressed into a smaller size, and if its using encryption mode then this compressed data is encrypted.

The header of these files determines the mode used. This is a common occurrence in Telltale files. These are magic number strings (covered in Section 1: *Terminology*) and have 3 possible values:

- TTCN (TellTale Container Normal(?)),
- TTCz (TellTale Container Compressed),
- TTCe (TellTale Container Encrypted - & Compressed).

The first one is the uncompressed mode. This mode consists of just the 4 byte string followed by an 8 byte (long integer), which is the size of the payload data. In a tabular form:

TTCN/ID	Telltale Data Container: Normal Mode	Type	Sign
00	Magic Number	M (TTCN)	N/A
01	Payload Data Size	8	U
02	Payload Data	Var[/01]	N/A

The next two modes are identical in format, however the encryption mode just means when the data is read in, before it is decompressed it needs to be decrypted. I will refer to these modes as just compressed from here on.

The compressed format is described in the following table:

TTC/ID	Telltale Data Container: Compressed Mode	Type	Sign
00	Magic Number	M (TTCe or TTCz)	N/A
01	Compression Number	4	U
02	Compression Window Size	4	U
03	Compressed Chunk #	4	U
04	Compressed Chunk Offsets	8[/03 + 1]	U
05	Compressed Chunks	Var[...]	N/A

There are also two other magic numbers which exist to simplify the headers:

- TTCZ,
- TTCE.

These, with capitals, are the exact same except they default to compression type *Z-lib*. This means that they skip variable 01 in the above format. These are the most common.

There are two types of compression which Telltale uses:

- Z-Lib v1.8.0, raw deflate (nowrap = true).
- Oodle LZA.

Oodle compression is used only a handful of times and is only used in very large game archives. Oodle is proprietary compression engine and therefore may be hard to figure out, but you can use C++ to hook into the DLLs (at least on windows) to do this.

The values for compression type are below:

Compression Number (TTC/01)	Compression
0	Z-Lib
1	Oodle LZA

In the compressed format table, we then have the window size. This window size is the decompressed size of each of the compressed chunks which is passed to the decompressor. You can find the total size of the uncompressed data by multiplying this window size by the number of chunks. When an data container is serialised, the total size of the input data is padded with zeros (zeros are added to whatever the size is) so that the the input data size is a multiple of window size.

We then have the compressed offsets. These are the offsets in the file (or data stream) *relative to the magic number offset*. As this is a data container, it *wraps* the input data to make its size smaller. The magic number may not be at offset 0, if, for example this data container is embedded in another file. So ensure if you are reading from a file then you add the base offset of the magic number to all of these offsets.

These offsets are all 64 bit long integers which are unsigned. Notice that there are number of chunks plus one offsets stored. The last offset is the offset of the end of the file (so the total file size in theory - of the data container). This is needed so you can calculate the size of the last compressed chunk:

$$S_X = \text{Offsets}[X+1] - \text{Offsets}[X]$$

Where S_X is the size of chunk X.

Reading Data

Suppose you want to read 10 bytes from offset 295 in a data container. Assume in this example that the window size is 100 (it is pretty much always - in practice - 65536 bytes (0x10000 bytes)). We can find the first chunk index by dividing the offset by the window size, and ignoring the remainder: resulting in chunk 2 (295 integer divided by 100). We then need to use the offset array read in the file to find the offset we need to go to, so that we can read in the compressed chunk. From that offset, we read the compressed chunk into a temporary buffer (buffer is just a store of bytes of fixed length), using the equation above to find the size of the compressed chunk which needs to be read in. If the data container is encrypted, we then decrypt it here. After decryption (remember decryption just 'untangles' the bytes but doesn't change the size), then we decompress the compressed chunk (which is in theory is less than 100 bytes - as it is compressed). Now we have the uncompressed data. We are at offset 95 in this chunk (chunk 2, and want to read from offset 295 in the file so 95 in this chunk - chunk size is 100). We can see we have 5 bytes which we can read from this chunk so we read then into the output buffer which we are reading to. Notice now we reached the end of the current chunk, so we now need to get the next chunk by repeating the process above. After that, you have chunk 3 and we can copy the first 5 bytes into the last 5 bytes of the output buffer, leaving us at offset 5 in chunk 3.

Below is example Java code of this happening:

```
long chunkoff = mPosition % mWindowSize;
if(mCachedChunkIndex != chunkoff)
    GetPage(chunkoff);
long rem = mWindowSize - chunkoff;
if (rem >= size) {
    System.arraycopy(mCachedChunk, (int)chunkoff, b, off, size);
    mPosition += size;
    return;
}
byte[] out = b;
int out_idx = off;
if (rem!=0) {
    System.arraycopy(mCachedChunk, (int)chunkoff, out, out_idx, (int)rem);
    out_idx += rem;
    size -= rem;
}
mPosition += rem;
if (mWindowSize >= size) {
    GetChunk(mCachedChunkIndex + 1);
    System.arraycopy(mCachedChunk, 0, out, out_idx, size);
}
else {
    int blocks = size / mWindowSize;
    for (int i = 0; i < blocks; i++) {
        GetChunk(mCachedChunkIndex + 1);
        System.arraycopy(mCachedChunk, 0, out, out_idx+i*mWindowSize, mWindowSize);
    }
    rem = size % mWindowSize;
    if(rem!=0) {
```

```
            GetChunk(mCachedChunkIndex + 1);
            System.arraycopy(mCachedChunk, 0, out, out_idx+blocks*mWindowSize, (int)rem);
        }
    }
```

We use cached chunk index and cached chunk state variables to hold the index of the currently decompressed page in memory. The position member variable holds the current position in the whole data container, where offset 0 is the beginning of the decompressed data. The `GetChunk` function decompresses the given chunk index into cached chunk and sets cached chunk index to that index passed in.

When writing data containers, I advise you to write the compressed chunks to a temp file and then use the sizes there to write first the offset array and then just transfer the temp file data into the final output data stream.

Use Cases

This data stream container format is used mainly for TTArchive2 bundles (.TTARCH2 files) which are similar to zip and just bundle up all the files into one file which is a lot better than having a folder with thousands of files. Another case is that telltale shader pack files (.T3FXPACK) use these to compress shaders which are very repetitive, especially with thousands similar shader programs. Furthermore save data and event bundles (.BUNDLE, .ESTORE, .EPAGE files) use this. Meta Stream files all have the capability to store a container instead of the raw data. This is described in the Meta stream section.

3: Encryption

Telltale throughout its lifetime has used a small few methods of encryption. Pretty much all of its encryption makes use of the *Blowfish* algorithm. This cipher uses cipher keys (encryption keys) of length 32 to 448 bits (4 to 56 bytes). All of Telltale's encryption keys used are 55 bytes long and are all found as upper case hexadecimal strings with their respective game ID in *Appendix B*. In games newer than and including *The Wolf Among Us [52/*]* a modified version of the algorithm is used. The difference in the modified algorithm is very small and I will cover it here so you can implement it (by copying Blowfish code and changing it).

Introduction to Block Ciphers

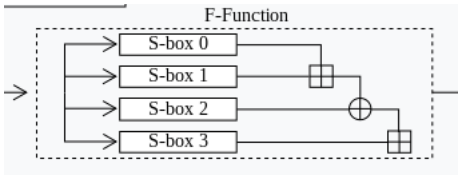
Here I will give a quick introduction to block ciphers so that even if you are not familiar with how encryption works you can still have an idea on how you implement you own data container parsers and synthesisers while still having an idea of what you are doing and what Telltale did to modify the cipher. Data containers are the only format which makes use of this encryption in newer games, and in older games the game archives do.

A block cipher (cipher meaning a way of transforming data) is a cipher that just works in blocks. Blowfish is an example of this and the block size is fixed at 64 bits. This means that long integers (64 bit) are encrypted one by one 'independently' of each other. The blowfish block cipher uses what is called an SP-network, or a substitution-permutation network. When we initialise the cipher with a given encryption key, each byte from the key is used to generate the S-box (substitution box) and P-box (permutation box). These are both arrays of 64 bit integers (in ciphers these are always unsigned), and are both key dependent.

S-boxes and P-boxes both transform input data (bits) into output data. S-boxes *substitute* a smaller number of bits for another small number of bits, hence the name. This is always one-to-one, meaning that each set of input bits creates one distinct set of output bits, allowing for this process to easily be reversed in decryption. The P-boxes *permute* the data bits, which means changing the order of them and rearranging them, hence the name.

Blowfish also uses what is called a *Feistel* network. This is an algorithm which actually encrypts each block of data and this is done some fixed number of times, called the number of *rounds*. This algorithm splits the input block (in this case 64 bits) into two sub-blocks, which normally are of equal size. In Blowfish they are equal size, so we have two 32 bit sub-blocks which we call left and right, left being the top 32 bits of the 64 bit block and right being the bottom two 32 bits (that is, in big endian. In little endian this is switched as the first four bytes in memory here are the bottom 32 bits). Each round, firstly the left sub-block is XOR'ed (a bitwise operation explained below) with a sub-key unique to that round - in Blowfish this is the value in the P-box at the index equal to the current round (rounds 0 through 15 - 16 in total). This value then does two things. First it becomes the *right* sub-block in the next round, and then it also goes through a function *F* which generates a

unique other 32 bit sub-block of different value using the key. In blowfish, this function is constant and is defined in the following image:



Source: Wikipedia

There are 4 S-boxes in Blowfish, each comprising of 256 32 bit unsigned integers. The 4 inputs in the image above are the 4 quartets (bytes, 8 bits) from the input 32 bits of the sub-block. The lowest 8 bits are the index into S-box 0 and the highest 8 bits are the index into S-box 3, and same follows for the middle 2 quartets. The square with a addition symbol inside means the two are modulo-32 added. This just means the two 32 bit unsigned values are added together and we ignore any overflow bits on the result after the addition (overflow meaning any bits that are non-zero which lie above the thirty second bit). The circle with an addition sign just means XOR, a bitwise operation in which the output bit is only 1 if the two input bits are not the same (0 XOR 0 and 1 XOR 1 both equal 0, however 1 XOR 0 and 0 XOR 1 both equal 1). XOR stands for exclusive OR, a version of OR but only true when they are not equal as well.

A bitwise operation is an operation which takes two input *bits* (0 or 1) and returns an output bit. When performing a bitwise operation on two X bit integers, each of the X bits in both inputs undergo the operation with the corresponding bit with each other. For example, bit 5 of the first input integer will go under the operation with bit 5 of the second input integer at bit index 5 - the same goes for all the bits in the two input values.

After the 32 bit value has gone through this function *F*, it is XORed with the right sub-block in this round. This new value becomes the left sub-block for the next round. This is repeated 16 times, 16 being the number of rounds in a blowfish cipher.

What makes this algorithm very useful is that when you pass the encrypted data right into this algorithm again, you get the original unencrypted data. The only thing needed to change between encryption and decryption is the index in the P-box: in decryption the index should start from 15 and go to 0. The reason this all can work is because XOR has a special property that it undoes itself: the reverse of an XOR operation is itself.

After these rounds are completed, blowfish has two extra steps: undo the last swap of round 15 (15 is the last round, 0 being the first). This undo is done by just swapping them again. The P-box for Blowfish is 18 bytes long. The first 16 bytes have been used in each round. The last 2 bytes are used here. We just XOR the final two sub-blocks, left and right, with P-box values at indices 16 and 17 respectively. These last two steps are known as *output whitening.* Of course, we then need to undo this when we decrypt - so when we do the loop of rounds, instead of starting from index 15 to 0, we go from 17 to 2 and then after the loop we just XOR the final two sub-blocks with the indices 0 and 1 of the P-box. In this algorithm we then finally just combine the two sub-blocks again to get the output 64 bit block which is either the encrypted data in encryption or decrypted data in decryption.

The last part of the algorithm is the *key schedule.* This is the part of the algorithm which initialises the S and P boxes, using the input encryption key. This is the initialisation part of the algorithm and happens before encryption/decryption can happen. Blowfish's key schedule first initialises all of the boxes to a set of constants which are derived from the hexadecimal digits of the fractional part of pi. This is because they have no obvious pattern - these types of numbers are referred to as *nothing up my sleeve numbers*. These are constant and can be found on the internet. Then, using the key, each value of the P-box is xor-assigned (the value is set to the current value XOR'ed with the argument) with the key at that index (the argument). Remember that the P-box values (18 of them) are unsigned 32 bit values. Since the key is a string of bytes, we will need to XOR 4 bytes of the key with each P-box value since there are 4 bytes the integer type. Doing this we need to know the endian of the value we XOR with from those 4 key bytes. This is traditionally always *big-endian* in most encryption algorithms - including Blowfish (MD5 and some others use little). Remember also that Blowfish keys are 32 to 448 bits long (4 to 56 bytes or 1 to 14 integers). We know there are 18 32 bit P-box values which need to be XOR'ed with, so we don't have 'enough' of the key to work with. Since the P-box is initialised with random digits from pi - we can just start using bytes from the beginning of the key when we reach the end of it. To achieve this, we can modulo the index from the

key array with the length of the key in bytes to make it wrap around when needed. Modulo is the remainder operation, similar to divide.

Telltale Blowfish Implementation

As mentioned earlier, all newer games use a modified version of this encryption. The two differences in the algorithm are very small.

Modification 1: S-box constants

The first modification one of the values in the S-box is endian flipped. This means that the bytes are flipped to the opposite endian: meaning the lowest byte in the 32 bit value becomes the top byte, and the middle two bytes swap. *The only value this applies to is in the first S-box (S-box index 0) at index 118*.

Modification 2: P-box indices

Recall that in each round of the encryption we XOR the left sub-block with the P-box at the index of the current round. In this modification, instead of accessing indices 1 to 4 directly, we map the indices in the P-box at this stage, changing the index with the following mappings:

Round Number (Original P-Box index)	Modified P-Box index
1	3
2	4
3	1
4	2
Any Other Index	Normal Round Number

In the decryption, we do something very similar but reversed - since we want to undo the encryption:

Round Number (Original P-Box index)	Modified P-Box Index
4	2
3	1
2	4
Any Other Index	Normal Round Number

Notice that round number 1 is not included here. This is because in the decryption, we loop from indices 17 to 2 (both inclusive). So the value 1 would never be reached. We loop excluding 1 in the normal algorithm anyway because we need to start on index 17 to undo the output whitening - and we need only do 16 rounds because thats how many we did to encrypt. In the decryption (and in normal Blowfish) we need to then finally XOR both left and right sub-blocks with P-box indices 1 and 0 (right and left respectively). This will undo the first two rounds of encryption of XOR with P-Box indices 0 and 1, creating the original input data. However since in the modified version we are switching round number 1 with P-Box index 3, in the final 'un' whitening stage in decryption we instead of XOR-ing the two sub-blocks with indices 1 and 0 we use indices 3 and 0 since in the table round number 1 corresponds to P-Box index 3.

That is all that Telltale modified in the encryption, we you will find the decryption works perfectly when you apply these two changes in newer games. In *Appendix C* I give an implementation of the key schedule (where you initialise the algorithm with the encryption key) as well as the encryption and decryption, showing the original and modified versions of the algorithm.

Additional Note

Note that when you initialise the key in both algorithms, you should use the normal unmodified Blowfish to encrypt the P-box. In the modified version initialiser, ensure you do this otherwise you will be using the wrong P-box values!

4: Archive Format (Version 2)

In this section I will outline the file format for Telltale Archives (game data bundles) - specifically for the new version of them called *TTArchive2's*. The file extension for these files is *ttarch2* and these contain all game data and files which the

game engine loads in games newer than and including *The Wolf Among Us [52/*]*. These files are found inside the game directory, in the sub-folder called *Archives*.

The version two here should not get confused with the versions mentioned below. The format in this section *as a whole* is the second revision of Telltale game data bundles - as opposed to version one (legacy version) being the previous file format with the extension of just *ttarch*.

Note that all telltale archives are wrapped in a data container (see *Section 3*). So you will need to use that to read from the file, with the following format as the payload data in it.

There are three versions for these files and they are very similar. The magic numbers for this format are the following:

- TTA2,
- TTA3,
- TTA4.

The TTA means TellTale Archive, Version XXX where XXX is the number after the TTA. The format of archives is the following:

ID	Telltale Archive V2 [TTA]	Type	Sign
00	Magic Number	M (TTA2/TTA3/TTA4)	N/A
01	Unused	4	U
02	Name Table Size	4	U
03	Resource Count	4	U
04	Resource Info Table	F[TTAResource][/03]	N/A
05	Name Table	Var[/02]	N/A
06	Resource Data	Var[EOF]	N/A

The 'Unused' variable (ID 01) in the archive format above is only present if the archive version is 2 or 3 (TTA2 or TTA3). This value tends to be 0, but when you read files you should ensure that the unsigned value is between 0 and 15 inclusive. Values of 16 or higher signify the archive is corrupted.

The resource info table contains is a list of the resources information such as their size, offset and name table offset. That format is below:

ID	Telltale Archive V2 Resource [TTAResource]	Type	Sign
00	File Name Hash	8	U
01	File Offset	8	U
02	Unused 1	4	U
03	File Size	4	U
04	Unused 2	4	U
05	Name Page Index	2	U
06	Name Page Offset	2	U

There are two unused integers in this format. These can be ignored and should always be 0. Unused 1 is *only* present when the archive version is 2 (TTA2). Unused 2 is *always* present. The file offsets in the archive are all relative to the offset starting at the first byte of the resource data after the name table.

The name table is a large buffer of raw 1 byte ASCII text, with null terminators between them, which contains the actual strings of all of the file names. The name table is separated into pages of equal size - with the size being fixed at 65536 (0x10000) bytes. The name table size integer in the TTA format should always be a multiple of this value. The number of name pages can be derived from that value, by dividing it by 65536. The name tables are created by just continually adding string text to the name table buffer - and when all have been added for all files, this buffer is padded to a length of the next multiple of 65536 bytes.

Each resource in the TTAResource format has a name table index and offset. The name table index refers to the index of the name tables which the file name is found in. The name table offset is the offset into that name table at which the file name string starts. Note that if you just read the name table into one big buffer, you can calculate the offset of the filename into that buffer by just shifting the name table index into the upper 16 bits of the 32 bit unsigned offset variable you are using, and have then having the bottom 16 bits being the name table offset.

It is useful to note that in the actual Telltale implementation of this format, there should be no more than 4096 name tables (4096 * 65536 byte name table size max) - and there should also be no more than 65535 files in one archive. If any of these conditions are not met then engine will not load the archive - it will be signalled as corrupt.

Below is an example of some Java code which reads an archive, where `mInputContainer` is data container which the data is read from:

```
String magic = ByteUtils.ReadString(mInputContainer,4);
boolean bExtra = false;
if((bExtra=(magic.equals("2ATT"))) || magic.equals("3ATT")) {
    int unk = ByteUtils.ReadInt(mInputContainer);
    if(unk > 15)
        return false;
}else if(!magic.equals("4ATT")) {
    Telltale.Error("Could not open TTArchive2 as header was invalid");
    return false;
}
int name_pages = ByteUtils.ReadInt(mInputContainer);
int resources = ByteUtils.ReadInt(mInputContainer);
if(name_pages > 0x10000000 || resources > 0xFFFFF) {
    Telltale.Error("Archive corrupt");
    return false;
}
mResources.clear();
mResources.ensureCapacity(resources);
for(int i = 0; i < resources; i++) {
    Resource resource = new Resource();
    resource.mSymbolCrc = ByteUtils.ReadLong(mInputContainer);//Filename hash
    resource.mOffset = ByteUtils.ReadLong(mInputContainer);
    if(bExtra)
        mInputContainer.Read(4);//Unused 1
    resource.mSize = ByteUtils.ReadInt(mInputContainer);
    mInputContainer.Read(4);//Unused 2
    int nblk = ByteUtils.ReadShort(mInputContainer);
    int nidx = ByteUtils.ReadShort(mInputContainer);
    resource.mNamesOffset = nidx | (nblk << 16);
    mResources.add(resource);
}
mNamesTable = mInputContainer.Read(name_pages);
mFileStart = mInputContainer.GetPosition();
```

5: Script Format

In this section I will cover the file format for compiled and encrypted *Lua* scripts which are used in the game engine. I will cover both with extension of normal .lua as well as the legacy .lenc (lua encrypted) files.

(Newer) Lua Format

Newer games scrapped the old .lenc script format. Lua files now use the normal lua extension but the files are encrypted. The files are not, however, just encrypted as a whole. There 3 magic numbers you should look out for when parsing a Telltale script:

- [0x1B]Lua (0x61754C1B) [Compiled Lua]
- [0x1B]LEn (0x6E454C1B) [Compiled Encrypted Lua]
- [0x1B]LEo (0x6F454C1B) [Compiled Encrypted Lua]

Note that the [0x1B] in the string version means the hex value of 1B (in normal base 10 this is 16(1) + 11(B) = 27), which is not a renderable ASCII character - it is the escape character. In brackets I put the little endian unsigned 32 bit hex value for the magic numbers. In square brackets is what type of Lua script it is. The 'LEn' part presumably stands for Lua ENcrypted. The last one is very similar.

If you encounter the first header, then you are dealing with a compiled Lua script. This means you can just run it, or decompile it to your liking. The escape character being part of this magic number allows for you to be able to test if it is actual Lua code or is compiled code, as Lua code will not have the escape character as it not a alphanumerical character.

However if you encounter the 'LEn' header, then you will need to be Blowfish decrypt the *rest* of the file, not including that 4 byte magic number. The decrypted data will *not* contain the actual compiled Lua magic number, which is [0x1B]Lua. Simply read in the whole file, decrypt from offset 4 until the end of the file, then finally replace the first 4 bytes from the [0x1B]LEn header to the [0x1B]Lua header.

If you encounter the 'LEo' header then decrypt the rest of the file, offset 4 an onwards, and remove the 'LEo' part. The encrypted data will contain the actual compiled Lua header mentioned above - so the output file size will be 4 bytes smaller, as opposed to the same in the 'LEn' version.

Note that you only need to decrypt these scripts. If you decrypt the script, optionally decompile, optionally recompile it (in the correct version and endianness of course) then the game engine will be able to read in the file. You don't even need to recompile it, plain source files will load. It is better to compile of course, so I would recommend you do. But there is no need to re-encrypt the file. If we look at the decompiled game engine code below, you can see we can do this because of some code in the `ScriptManager::DoLoad` function:

```
244  result_1 = lua_load(
245               L,
246               (const char *(__cdecl *)(lua_State *, void *, unsigned __int64 *))LoadInfo::Load,
247               &pCodeBuffer,
248               chunkname_1,
249               "bt");
```

The last argument to the function, string 'bt', is the mode. This stands for binary and text, allowing for both precompiled and just text code to be loaded into the Lua runtime.

For those interested, below shows some of the code which parses the encrypted headers:

```
165  bRemovePrevMagic = size >= 4
166               && (*(_DWORD *)v53.mpData == *(_DWORD *)"\x1BLua" || *(_DWORD *)v53.mpData == *(_DWORD *)"\x1BLEn");
167  bEncrypted = size >= 4
168         && (*(_DWORD *)v53.mpData == *(_DWORD *)"\x1BLEo" || *(_DWORD *)v53.mpData == *(_DWORD *)"\x1BLEn");
```

It is also noteworthy, that the 'LEo' header by default is only used for *resource descriptors*, that is the *_resdesc* files (this is the file prefix) which are found in the game Archives folder along with all of the game archives. These files, when decrypted, by default contain uncompiled Lua code. Although, the exception here is *_rescdesc* files, notice the 'c' between the 'res' and 'desc', which are not encrypted or compiled and just update build version information - these files can be ignored.

(Older) Lenc format

.lenc files (Lua encrypted files) are very simple and just need to be fully decrypted as a whole to get the compiled Lua code. These were used in older games, and before these files it was normal Lua files which were not encrypted but were found in game archives which may have been encrypted.

In most games before *The Wolf Among Us*, lua and *.lenc* files are used as scripts. These are either just raw script source code, compiled Lua code or encrypted. To tell, you have to read the first 4 bytes of the file. Telltale encrypts their data not just by blowfish in these older games. They use encrypt every so often in blocks, and sometimes just XOR with -1, meaning the bits are just flipped. See the game data container section for more information above.

6: Soundbank Format

This section covers the sound bank format which is used in most newer Telltale games. The file extension for these files is .FSB5 - which isn't actually a Telltale file format. This is a proprietary file format developed by the sound library which the Telltale engine uses - fmod. This format is not well documented and I will document it here. These files tends to be 'wrapped' into a .BANK file. This bank file contains a very long header in the *RIFF* file structure which is well documented but it does not give the actual format which we can interpret - so I will show a fast way to skip the large header and get straight to the .FSB5 file which is right after it. It may be that the .BANK file does not contain an FSB5 file (in Telltale these .BANK files tend to be prefixed with _strings) - in this case you won't find anything of particular use in these files. Note this format is in little endian.

RIFF Format Structure

This part only applies to .BANK files, and not FSB5. The FSB5 is found in a chunk inside this format, which is described later.

RIFF files (Resource Interchange File Format) were developed my Microsoft, and are used in many places such as .WAV files and .AVI files. Here, it is used to carry lots of information about the sound bank. The RIFF format uses a chunked approach: each chunk consists of a 4 byte ASCII string ID, a little endian 32 bit unsigned integer which is its size in bytes and then the chunk data. This chunk data is padded (zeros are appended to the data) such that the last byte in chunk data lies at an even offset in the file.

The start of the file should always contain the ASCII string 'RIFF' which is the definition of the first chunk. The chunk ID can be whatever you want, and is designed so that it can be custom. The only reserved ones are RIFF and LIST. RIFF is the root chunk, and LIST chunks are just a list of chunks in which the number of chunks in the LIST chunk is determined when you read them from the file, until the number of bytes remaining in the LIST chunk is 0. The RIFF chunk is a LIST chunk.

Below is a simple, easy to read, Java implementation of the parser of a RIFF file:

```
public class RIFF {

    // CLASS MEMBERS & TYPE DEFINTIONS

    public abstract class Chunk {
        public String ID = "";
    }

    public class DataChunk extends Chunk {
        public byte[] Data = null;
    }

    public class ListChunk extends Chunk {
        public String TypeID = "";
        public ArrayList<Chunk> SubChunks = new ArrayList<RIFF.Chunk>();
    }

    public ListChunk RIFF;

    // READ

    public boolean ReadIn(RandomAccessStream in) throws Exception {
```

```
        boolean bSuccess = (RIFF = (ListChunk)ReadChunk(in, in.getFilePointer())) != null;
        if(bSuccess&&!RIFF.ID.equals("RIFF")) {
            System.out.println("Root chunk is not 'RIFF'");
            bSuccess = false;
        }
        return bSuccess;
    }

    ListChunk ReadLIST(RandomAccessStream in, int size, String ID, int fileStart) throws Excepti
        int start = in.getFilePointer();

        ListChunk c = new ListChunk();
        c.ID = ID;
        c.TypeID = new String(in.readNBytes(4), StandardCharsets.UTF_8);

        if(size > 4) {//4 because the first 4 bytes in this chunk are the TypeID above.
            int remaining=1;
            while(remaining > 0) {
                Chunk Sub = ReadChunk(in, fileStart);
                if(Sub == null)
                    return null;

                remaining = size - (in.getFilePointer() - start);
                if(remaining < 0) {
                    System.out.println("Corrupt RIFF");
                    return null;
                }

                c.SubChunks.add(Sub);
            }
        }

        return c;
    }

    Chunk ReadChunk(RandomAccessStream in, int fileStart) throws Exception {
        String ID = new String(in.readNBytes(4), StandardCharsets.UTF_8);
        int size = ByteUtils.ReadInt(in);

        if(ID.equals("RIFF") || ID.equals("LIST"))
            return ReadLIST(in, size, ID, fileStart);

        DataChunk data = new DataChunk();
        data.ID = ID;
        data.Data = in.readNBytes((int)size);

        //If needed read pad byte
        if(((in.getFilePointer() - fileStart) & 1) == 1)
            in.read();

        return data;
    }
```

```
}
```

Finding the FSB Chunk

The FMOD sound bank which contains all of the sound files (the .FSB file) is embedded into some .BANK files. If it is embedded, there will be a chunk at the end of the file which has the ID 'SND ' with a space after SND - obviously meaning 'Sound'.

FMOD SoundBank Format V5

Here I go over the format for FSB5 (FSB version 5) files. These come under the .FSB file extension which can be extracted from some .BANK files. If the version is 5, then the format is described below.

Header Block

ID	FMOD Sound Bank [FSB5]	Type	Sign
00	FSB5 Magic	M(FSB5)	N/A
01	Version	4	U
02	Number of Audio Files	4	U
03	Header Size	4	U
04	File Name Table Size	4	U
05	Total Audio Data Size	4	U
06	Audio Format	4	U
07	Unknown Value A	4	U
08	Unknown Value B	4	U
09	Unknown Value C	8	U
10	Unknown GUID	16	U

If the version is not 1, then the following does not apply. However it is likely very similar and may just have some extra or removed data which is comparable to the format described here.

As you can see there are a few unknown values towards the end of the header. Unknown value A is normally always 1, and B is normally always 0. Unknown value C can either be 0 are some random non-float data. Unknown GUID is likely the bank GUID but thats an educated guess. These variables you should just read, then write back exactly what you read when opening the file.

The table below lists all the known values for (ID 06) audio format:

Audio Format Value	Audio Format
01	PCM8 (8 bit wav)
02	PCM16 (16 bit wav)
03	PCM24 (24 bit wav)
04	PCM32 (32 bit wav)
05	PCM Float (32 bit wav)
15	Ogg Vorbis (.ogg)

File Information Block

After this header block, we have the file information block. Here for each audio file we start by reading an unsigned 64 bit bitfield. Depending on this bitfield, there are 0 or more unsigned 32 bit extra information integers which give more information about the audio file.

The following table gives the ranges of the bit field and the data that they represent:

ID	0 Indexed Bit Range (Inclusive-Inclusive)	Meaning
0	[00, 00]	Bool. Has next 32 bit chunk info.
1	[01, 04]	Audio File Frequency
2	[05, 05]	Channel Num Minus One
3	[06, 33]	Audio Data Offset
4	[34, 63]	Audio File Sample Count

Audio File Frequency is a 4 bit integer nibble which maps to the following frequencies:

Serialised Nibble	Frequency in Hz
1	8k
2	11k
3	11,025
4	16k
5	22,050
6	24k
7	32k
8	44,100
9	48k

The number of channels is stored as a boolean (ID 2), where 0 means 1 channel (mono) and 1 means 2 channels (stereo).

Audio Data Offset is relative to the start of the audio data block which starts after the name table block, 0 meaning the first byte after the name table block.

The number of samples is the number of sampled points on the full audio waveform.

If the LSB is set (ID 0), then we have another 32 bit unsigned int block after:

ID	0 Indexed Bit Range (Inclusive-Inclusive)	Meaning
01	[00, 00]	Bool. Has another 32 bit chunk.
02	[01, 24]	Chunk Size in Bytes
03	[25, 31]	Chunk Type

Following this unsigned 32 bit value, is (ID 02) number of bytes - the chunk data. You can skip this if the chunk type is not one of the following. If the LSB (ID 01) is non zero, there is another 32 bit unsigned value chunk info which you need to read, so you should repeat this.

The following table lists the known chunk types (ID 03):

Chunk Type ID	Meaning	Chunk Data
01	Number of Channels Override	1 Bool
02	Frequency Index into Table Above	1 32 bit int
...	Unknown Chunks that can be skipped	Chunk Size in Bytes bytes
11	Vorbis (.OGG) Info	Header CRC32, followed by other stuff

If the Vorbis chunk is present then the first 4 bytes is the CRC32 hash of the files common .OGG file header. Simon Pinfold has these on Github under his *python-fsb5* repo. The rest of the data is unknown so you should skip it.

Name Table Block

After all of the file information has been read there is the name table block. Firstly there are all of the file name offsets. For each audio file read one unsigned 32 integer which is the offset relative to the name table strings buffer which immediately follows this buffer. Note that to find the size in bytes of the name table strings buffer after the offset indices, you just take 4 times the number of audio files from the name table block size which is in the main FMOD sound bank header. These strings are all null terminated with a 0 byte.

Audio Data Block

Finally there is the actual audio waveform data. All file data offsets from the file information block above are relative to the offset at the start of this block, right after the last byte in the file name table.

PCM Audio Data Format

For PCM audio, the final frame data is just the full audio data block.

Vorbis Audio Data Format

For ogg Vorbis, the data is a list of audio packets. First, read an unsigned short from the audio data. This is the size of the packet, which follows directly after. If this unsigned short is zero, then this is the end of the audio file data. This translates directly to a Vorbis packet. You may need to use the ogg and Vorbis C libraries to write these packets accordingly. The first packet in the final output ogg file should be the ID header, followed by the comment packet and then finally the setup header of which its bytes come directly from the CRC32 stored in the Vorbis information chunk. After these three packets, you should write the audio data packets. More information on the ogg Vorbis format can be found at Xiph's official documentation here.

Appendices

Appendix A:

C++ CRC64 Implementation

```
//ECMA-182 CRC Table, can be found online.
extern const u64 crc_tab[256];

// Calculate the 64 bit CRC of a given data buffer of length len.
inline u64 CRC64(const char* const buf, u32 len, u64 crc = /*initial CRC64*/0) {
    for(int i = 0; i < len; i++){
        crc = crc_tab[((int)(crc >> 56) ^ buf[i]) & 0xFF] ^ (crc << 8);
    }
    return crc;
}

//Implement CRC64_CaseInsensitive yourself... (maybe pass in a string
//instead of a buffer!)
```

Appendix B:

Game Encryption Keys

N/A meaning the game does not use encryption.

Game ID	Modified Encryption (Y/N)	Hexadecimal Encryption Key String
00	N/A	N/A
01	N	82A3898889D89FB7D3D8DAC082D7C2C1CE8DA1EA99B7A5DDCA52E58769C8A46EBB9997BBCDD79
02	N/A	N/A
03	N	81D89B9956E26573B4DBE3C964DB8587AB999BDC6FEB689FA790DDBA6BE29364A1B4A0B493D96B
04	N	92CA9A8185E46473A3BFD6D17FC6CB88995B80D8AAC297E79651A0A89AD9AE95D7766280B4C4A6
05	N	92CA9A8185E46473A4BFD6D17FC6CB88990180D8AAC297E79651A1A89AD9AE95D7766281B4C4A6B
06	N	92CA9A8185E46473A5BFD6D17FC6CB88995D80D8AAC297E79651A2A89AD9AE95D7766282B4C4A6
07	N	92CA9A8185E46473A6BFD6D17FC6CB88995E80D8AAC297E79651A3A89AD9AE95D7766283B4C4A6

08	N	92CA9A8185E46473A7BFD6D17FC6CB88995F80D8AAC297E79651A4A89AD9AE95D7766284B4C4A6
09	N	92CA9A8185E46473A8BFD6D17FC6CB88996080D8AAC297E79651A5A89AD9AE95D7766285B4C4A6
10	N	82BC766867BF7C77B5BFBE9875B89C8BAC7D76AB80C87FA3A874B8897CBFAA68A2987B83A4B682
11	N	92CA9A8185E46573A3BFD6D17FC6CB89995B80D8AAC297E79751A0A89AD9AE95D7776280B4C4A6
12	N	92CA9A8185E46573A4BFD6D17FC6CB89990180D8AAC297E79751A1A89AD9AE95D7776281B4C4A6B
13	N	92CA9A8185E46573A5BFD6D17FC6CB89995D80D8AAC297E79751A2A89AD9AE95D7776282B4C4A6
14	N	92CA9A8185E46573A6BFD6D17FC6CB89995E80D8AAC297E79751A3A89AD9AE95D7776283B4C4A6
15	N	92CA9A8185E46573A7BFD6D17FC6CB89995F80D8AAC297E79751A4A89AD9AE95D7776284B4C4A6
16	N	87D89A9997E094B5A39CA6ACA1D2B8CADD8B9FA86DA67EDED286E2C99ADE9264908DA1BCC6D6A
17	N	87D89A9997E094B5A39CA7ACA1D2B8CADD8B9FA86DA77EDED286E2C99ADE9264918DA1BCC6D6AD
18	N	87D89A9997E094B5A39CA8ACA1D2B8CADD8B9FA86DA87EDED286E2C99ADE9264928DA1BCC6D6A
19	N	87D89A9997E094B5A39CA9ACA1D2B8CADD8B9FA86DA97EDED286E2C99ADE9264938DA1BCC6D6A
20	N	87D89A9997E094B5A39CAAACA1D2B8CADD8B9FA86DAA7EDED286E2C99ADE9264948DA1BCC6D6A
21	N	96CA99A085CF988AE4DBE2CDA6968388C08B99E39ED89BB6D790DCBEAD9D9165B6A69EBBC2C69
22	N	96CA99A085CF988AE4DBE2CDA6968389C08B99E39ED89BB6D790DCBEAD9D9166B6A69EBBC2C69
23	N	96CA99A085CF988AE4DBE2CDA696838AC08B99E39ED89BB6D790DCBEAD9D9167B6A69EBBC2C69
24	N	96CA99A085CF988AE4DBE2CDA696838BC08B99E39ED89BB6D790DCBEAD9D9168B6A69EBBC2C69
25	N	8CD89B9F89E57CB6DECDE3C8639584A4D89898DCB6BEA9DBC68FD386699DAEA3CDB097C8AAD6
26	N	8CD89B9F89E57CB6DECDE3C8639585A4D89898DCB6BEA9DBC68FD386699EAEA3CDB097C8AAD6
27	N	8CD89B9F89E57CB6DECDE3C8639586A4D89898DCB6BEA9DBC68FD386699FAEA3CDB097C8AAD6
28	N	8CD89B9F89E57CB6DECDE3C8639587A4D89898DCB6BEA9DBC68FD38669A0AEA3CDB097C8AAD6
29	N	8CD89B9F89E57CB6DECDE3C8639588A4D89898DCB6BEA9DBC68FD38669A1AEA3CDB097C8AAD6
30	N	82BC7669549C8690D7DAEAA785AE8887997D7ADCABEA79C2AE569F858CB9C6A2D4888598969369
31	N	92CA9A8185E46673A3BFD6D17FC6CB8A995B80D8AAC297E79851A0A89AD9AE95D7786280B4C4A6
32	N	92CA9A8185E46673A4BFD6D17FC6CB8A990180D8AAC297E79851A1A89AD9AE95D7786281B4C4A6B
33	N	92CA9A8185E46673A5BFD6D17FC6CB8A995D80D8AAC297E79851A2A89AD9AE95D7786282B4C4A6
34	N	92CA9A8185E46673A6BFD6D17FC6CB8A995E80D8AAC297E79851A3A89AD9AE95D7786283B4C4A6
35	N	92CA9A8185E46673A7BFD6D17FC6CB8A995F80D8AAC297E79851A4A89AD9AE95D7786284B4C4A6
36	N	87CE90A893DE6473A3B4DAC7A6D4C588995B75DCA0E9A5E19651A09D9ECFD5A3D1766280A9C89C
37	N	87CE90A893DE6473A4B4DAC7A6D4C588990175DCA0E9A5E19651A19D9ECFD5A3D1766281A9C89CE
38	N	87CE90A893DE6473A5B4DAC7A6D4C588995D75DCA0E9A5E19651A29D9ECFD5A3D1766282A9C89C
39	N	86DB96978FD89874A29DBCD69BC8BEC3CE5B5DA884E79FD2D08DD486699DA8A6C8A89DBBC694
40	N	82BC766A549C7696BBA2A59475B89C8D995A70CA86AB669FA874B88B699CA487A87B627FA4B682
41	N	82CE999986DE9CB7EBBCE4CF97D796BCD58F8FE9A6E9AFBFD48CD4C77CD1CD99C1B79BC3DAB3A
42	N	81CA909F78DB87ABD7B2EAD8A7D7B888995B6FD8A0E08ADEB989D49BAEE0D6A6C4766280A3C49
43	N	81CA909F78DB87ABD7B2EAD8A7D7B88899016FD8A0E08ADEB989D49BAEE0D6A6C4766281A3C49C
44	N	81CA909F78DB87ABD7B2EAD8A7D7B888995D6FD8A0E08ADEB989D49BAEE0D6A6C4766282A3C49
45	N	81CA909F78DB87ABD7B2EAD8A7D7B888995E6FD8A0E08ADEB989D49BAEE0D6A6C4766283A3C49C
46	N	81CA909F78DB87ABD7B2EAD8A7D7B888995F6FD8A0E08ADEB989D49BAEE0D6A6C4766284A3C49C
47	N	86DB96978FD89874A29EBCD69BC8BEC3CE5B5DA984E79FD2D08DD486699EA8A6C8A89DBBC6946
48	N	89DE9F9597DF9CA6C2CDE7CF639583A1DE9C8EEAB0DE99BFC693DA86699CABA9D1A6A5C2CAC68
49	N	8BCAA47592D082B5D6D1E79562959FB8E06B9BDB8CE79AD4D7529F8585CDD875CDA981C1C5C8AE
50	N	96CA999F8DDA9A87D7CDD9956295AAB8D59596E5A4B99BD0C9529F8590CDCD9FC8B39993C6C4
51	N	82CE999986DE9CB7EBBCE4CF97D7859ACE9692D9AFDEAAE8B590DABAAB9EA499CBAA94C1CAD7E

52	Y	85CA8FA089DF6473A2B2D6C69ECAC688995A73D89FE19BE296519F9B9ACECD99D276627FA7C49BI
53	Y	96CA999F8DDA9A87D7CDD9966295AAB8D59596E5A4B99BD0C9539F8590CDCD9FC8B39993C6C4
54	Y	81D89F9889DE9FA4E0D0E895629595C6DB8E92E9A9D6A4D3D8529F857BDBD398C4B79EB0CFC7AC
55	Y	86CA9A9973D287ABE4DBE3C9A5968387B08B9ADC8CDB8AD7D790DDBAAC9D9164A6A69FB4B0C9
56	Y	8CD29B9987DE94A9E69DA5947FCEC1BCCC9C8EDDB1A6669FB28ADDBA9CDEC29AD376627FAECCA
57	Y	96CA999F8DDA9A87D7CDD9B1639583AECA9698E0ABDC7AD4C685BC86699CB895CBB09BBDC8A7
58	Y	81CAA1A185DA6473A2AED6D89FC6C188995A6FD8B1E297DD96519F979AE0CE95CD76627FA3C4ADD
59	Y	96CA999F8DDA9A87D7CDD9976295AAB8D59596E5A4B99BD0C9549F8590CDCD9FC8B39993C6C4
60	Y	86DE8EA688D594B1E59DA59479DAB4C9CD938EE5B0A6669FAC96D0C79DD5C2A2D276627FA8D89A
61	Y	8CD29B9987DE94A9E69EA5947FCEC1BCCC9C8EDDB1A7669FB28ADDBA9CDEC29AD377627FAECCA
62	Y	81CAA1A185DA6573A2AED6D89FC6C189995A6FD8B1E297DD97519F979AE0CE95CD77627FA3C4ADD
63	Y	96CA999F8DDA9A87D7CDD9986295AAB8D59596E5A4B99BD0C9559F8590CDCD9FC8B39993C6C4
64	Y	96CA999F8DDA9A87D7CDD9BB93D1BEC0D79171DC9ED98DD0D18CD8C3A0B0C695C39C93BBCCCC
65	Y	92CA9A8185E46473A2BFD6D17FC6CB88995A80D8AAC297E796519FA89AD9AE95D776627FB4C4A6E

Appendix C:

Blowfish Encryption Code Examples with Modified Algorithm

Key Schedule:

```
unsigned int S[4][256];
unsigned int P[18];

//Endian swap the given input int
unsigned int ByteSwap(unsigned int num) {
        return(((num & 0xff000000) >> 24) |
            ((num & 0x00ff0000) >> 8) |
            ((num & 0x0000ff00) << 8) |
            ((num & 0x000000ff) << 24));
}

void KeySchedule(char* Key, int keylen)
    {
        int i,j,k;
        unsigned int datal, datar;

        for (i = 0; i < 18; i++)
        {
            //bfp is the P-box constants array derived from pi.
            P[i] = (long)bfp[i];
        }
        for (i = 0; i < 256; i++)
        {
            //ks0,1,2,3 are the S-box constants which can be found on
            //Bruce Schneier's website, along with the P-box constants.
            S[0][i] = ks0[i];
            S[1][i] = ks1[i];
            S[2][i] = ks2[i];
            S[3][i] = ks3[i];
        }
```

```
        //Below line swaps the endian of S[0][118]. Only in modified version.
        S[0][118] = ByteSwap(S[0][118]);

        j = 0;
        for (i = 0; i < 18; ++i)
        {
            data = 0x00000000;
            for (k = 0; k < 4; ++k)
            {
                data = (data << 8) | (Key[j%keylen]&0xFFl);
                j++;
            }
            P[i] = P[i] ^ data;
        }

        datal.value = 0x00000000;
        datar.value = 0x00000000;

        for (i = 0; i < 18; i += 2)
        {
            beo(datal, datar);

            P[i] = datal.value;
            P[i + 1] = datar.value;
        }

        for (i = 0; i < 4; ++i)
        {
            for (j = 0; j < 256; j += 2)
            {

                beo(datal, datar);

                S[i][j] = datal.value;
                S[i][j + 1] = datar.value;
            }
        }
        return 0;
    }
```

Blowfish F Function:

```
unsigned int F(unsigned int x)
    {
        unsigned int a,b,c,d,y;
        d = (x >>  0) & 0xFF;
        c = (x >>  8) & 0xFF;
        b = (x >> 16) & 0xFF;
        a = (x >> 24) & 0xFF;
        //First add the S-box values at indices from the top two bytes
        y = S[0][(int) a] + S[1][(int)b];
        //XOR with S-box at index lower middle byte
        y = y ^ S[2][(int)c];
        //Finally add with S-box index lowest byte of x
```

```
        y = y + S[3][(int)d];
        return y;
    }
```

Encryption:

```
//Use pointers so the input values are updated correctly.
void EncryptBlock(unsigned int* left, unsigned int* right)
    {
        unsigned int i, l, r, temp;
        l = *left;
        r = *right;
        //Loop from 0 to 15 (16 rounds).
        for (i = 0; i < 16; ++i)
        {
            //This switch statement is only in the modified version.
            switch (i) {
                case 1:  temp = P[3];   break;
                case 2:  temp = P[4];   break;
                case 3:  temp = P[1];   break;
                case 4:  temp = P[2];   break;
                default: temp = P[i];   break;
            }

            l ^= temp;
            r ^= F(l);

            //Swap left and right sub-blocks
            temp = l;
            l = r;
            r = temp;
        }

        //Ignore last swap by swapping again for whitening
        temp = l;
        l = r;
        r = temp;

        //Ouput whitening. Gets undone in first round of decryption.
        r ^= P[16];
        l ^= P[17];

        //Save changes
        *left = l;
        *right = r;
    }
```

Decryption:

```
//Use pointers so the input values are updated correctly.
void DecryptBlock(unsigned int* left, unsigned int* right)
{
        unsigned int i, l, r, temp;
        l = *left;
```

```
        r = *right;

        //Loop from 17 to 2. First two rounds undo encryption whitening
        for (i = 17; i > 1; --i)
        {
            //This switch statement only applies to the modified version.
            switch (i) {
                case 4:  temp = P[2];   break;
                case 3:  temp = P[1];   break;
                case 2:  temp = P[4];   break;
                //Case 1 is undone at the end in the output whitening, after last round.
                default: temp = P[i];   break;
            }

            l ^= temp;
            r ^= F(l);

            //Swap left and right sub-blocks
            temp = l;
            l = r;
            r = temp;
        }

        //Ignore last swap by swapping again for undoing the whitening
        temp = l;
        l = r;
        r = temp;

        //Ouput whitening. Undoes the first round of encryption
        r ^= P[1 /*This index should be 3 in the modified version*/];
        l ^= P[0];

        //Save changes
        *left = l;
        *right = r;
    }
```

To run:

```
void BlowfishEncrypt(unsigned char* data, unsigned int len){
        //Assuming key is set already...
        //Code it yourself, just go through the data in blocks of 8
        //And pass the left and right casted integers to the EncryptBlock
        //Ensure that endian is kept constant if data is used
        //across different endian machines here when casting to ints, eg
        //keep algo in little endian so if big endian call ByteSwap before calling
        //EncryptBlock. Do same in decryption.

        //Note that any leftover bytes if len is not a multiple of 8 will not
        //be encrypted.
}

//Same for decrypt but call DecryptBlock instead.
```