Share Files Securely

Justin Parr, 10/2013

Abstract

This document outlines a method for encoding, storing, and transmitting a file to one or more recipients securely by creating multiple bit streams that essentially contain no viable content, but by which the recipient can re-construct the original content.

This is achieved by using a symmetric operation to decompose a "high value" byte sequence (such as a file containing confidential data) in to blocks of random bit strings, and using hash "label" values to retrieve known bit blocks in order to reconstruct the original high-value byte sequence, based on instructions and label values contained in a "blueprint" string sequence.

Technical Assertions:

- A symmetric bitwise operation, such as XOR or EQV can be used to split a high-value bit sequence in to two random sequences that can be recombined in to the original sequence

- A hash function can be used to generate hash "label" values for a given bit sequence, that can be used as an index for storing and retrieving a specific bit sequence.

- Salt Data can be used to generate multiple, unique hash values for the same bit sequence, and can be used to generate multiple, unique index functions.

- A blueprint string containing hash values and assembly instructions can be used to locate, download, transform, and assemble various blocks of random bit strings in order to reconstruct the original high-value bit sequence.

- Using overlapping hash indices, multiple equivalent blueprints can be constructed that represent the same original bit sequence. This allows mutation and randomization when sharing blueprints.

Methodology

1. Demonstrate that a “high value” bitstream can be encoded as 2 or more random bit sequences plus a blueprint that can be leveraged to reconstruct the original high-value bitstream on the fly.

2. Provide a high-level algorithm for information-sharing using the above principle.

3. Review a legal analysis in the context of liability and plausible deniability with respect to encoding, storing, transmitting, or receiving “high risk” confidential content.

4. Review countermeasures that either complicate cryptanalysis or support plausible deniability.

5. Legal and Technical comparison of this method against other file transfer / file sharing mechanisms.

Symbols

Symbol	Definition
+	Boolean OR (returns “1” if either input is 1)
$\underline{+}$	Boolean XOR (returns “1” only if one input is 1 and the other is 0)
x	Boolean AND (returns “1” only if both inputs are 1)
$\underline{=}$	Equivalence function (returns “1” only if both inputs are the same)
/=	Not Equal
-	Inverse, -A = NOT(A) (returns “1” if input is 0, and “0” if input is 1)
f()	Function “f”
-f()	Inverse of function “f”, for example: -f(f(x))=x

The Associative Nature of the XOR Boolean Function

XOR is the “eXclusive OR” Boolean function, yielding 1 in the event that A and B inputs are different, and 0 in the event that A and B inputs are the same.

A	B	A $\underline{+}$ B
		XOR
0	0	0
0	1	1
1	0	1
1	1	0

The Equivalence function yields 1 where A and B are the same, and 0 where A and B are different, and is therefore equal to the negated output of A $\underline{+}$ B:

$A\underline{=}B = -(A\underline{+}B)$

Truth table:

A	B	A x B	A + B	A $\underline{+}$ B	A $\underline{=}$ B	-A
		AND	OR	XOR	EQU	NOT
0	0	0	0	0	1	1
0	1	0	1	1	0	▨
1	0	0	1	1	0	0
1	1	1	1	0	1	▨

When the output of XOR is combined with one of its inputs, XOR yields the OTHER input:

$A\underline{+}B = Z$,
$Z\underline{+}A = B$, and
$Z\underline{+}B = A$

A	B	Z = A $\underline{+}$ B	A = Z $\underline{+}$ B	B = Z $\underline{+}$ A
0	0	**0**	0	0
0	1	**1**	0	1
1	0	**1**	1	0
1	1	**0**	1	1

Since the Equivalence function is equal to the negated output of XOR, the Equivalence function exhibits the same associative property.

Decomposing A “High-Value” Bitstream

Given a “high value” bit stream, S, with a label of “Q01” (the labels will be discussed later), S might represent the bit sequence of some confidential data.

S = Significant (“High Value”) bit sequence

0	0	0	1	1	0	1	1	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	1	1	0	1	1	0	0	0

Label = Q01

F, whose label is “Q02”, represents a random factor, which is a random bit sequence with the same length of S

F = Random Factor

0	1	0	1	1	1	0	0	1	1	1	1	0	0	0	0	1	0	0	0	1	0	0	1	1	1	1	0	0	0	0	1

Label = Q02

Z, whose label is “Q03”, is calculated as S$\underline{+}$F, where XOR is applied in a bitwise fashion. For a given bit position “b”, Zb = Sb$\underline{+}$Fb

Z = XOR output

0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	0	0	1	0	0	1	1	1	0	0	1

Label = Q03

Because F is random, Z is therefore random.

Neither F nor Z contain S, due to the fact that both F and Z are random. Only F and Z together with the XOR function (combined) can re-create S – if any component (F, Z or XOR) is missing, S can’t be reconstructed.

Likewise, either F or Z *could* be XOR components of an entirely different bit stream, other than S.

Given Z, and another high-value bit sequence A:

1	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

There exists a bit sequence B that is also random, such that Z$\underline{+}$B = A

1	*0*	*1*	*1*	*1*	*0*	*0*	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*0*	*0*	*0*	*0*	*1*	*1*	*0*	*0*	*0*	*1*	*1*	*1*	*0*	*0*	*1*

Note that because F is random, Z is random, and therefore B is also random, and B can’t be related to S nor A.

Assuming that F is “really random” (not machine-pseudorandom), the relationship between Z and S will defy cryptanalysis, as it is as secure as a “one time pad” cipher, which, excluding human error or compromise, is considered to be “perfectly secure”.

The frequency with which Z or F are used to decompose other secure streams is directly relevant to the chance that enough information is available to an attacker, to mount an attack against S.

Using a bijective function to label bit sequences

A bijective formula has exactly one return value for each input value, and vice-versa. So for any given “x”, there is only one value of b, such that b=f(x), and likewise, only one x such that x= -f(b). For the purpose of this document, a bijective function can be implemented as a lookup table.

Using a bijective index function, q(), each unique bit sequence Bx can be labeled as Qx, where:

Qx=q(Bx)

Due to the bijective nature of q():

Bx=-q(q(Bx)

and

Qx=q(-q(Qx))

Using the sample bit streams listed previously, let's define the function, q() as follows:

String	B	q(B)
S (high-value)	00011011 00001111 11110000 11011000	Q01
F (random)	01011100 11110000 10001001 11100001	Q02
Z (XOR product)	01000111 11111111 01111001 00111001	Q03

For clarity,

q(00011011 00001111 11110000 11011000) = Q01

-q(Q01) = 00011011 00001111 11110000 11011000

q(-q(Q01)) = Q01

-q(q(00011011 00001111 11110000 11011000)) =
00011011 00001111 11110000 11011000

Since we use the variable “S” in place of the high-value bit stream:

S = 00011011 00001111 11110000 11011000

The above can be expressed more conveniently as:

q(S) = Q01
-q(Q01) = S
q(-q(Q01)) = Q01

$$-q(q(S)) = S$$

Since any bitwise operation, including XOR and EQU can operate on a bitstream, the relationship between S, F, and Z can be expressed as follows, using q():

$$-q(Q03) = -q(Q01) \pm -q(Q02)$$

Therefore:

$$Q03 = q(\ -q(Q01) \pm -q(Q02)\ )$$

Since XOR allows S, F, and Z to be interchanged, the following is also true:

$$Q01 = q(\ -q(Q03) \pm -q(Q02)\ )$$

In a practical implementation, q() would be a lookup table, and a unique “label” Qx would be generated using a one-way hash algorithm h(), such that for every value of “B”:

$$Qx = h(Bx)$$

One-way hash algorithms are not reversible, and would depend on q() and -q() (the index function) to find the original value of Bx.

From a practical perspective, h() can be implemented as any one-way hashing function (“secure” or otherwise) that is guaranteed to provide sufficient uniqueness to differentiate various bit streams, and includes MD5 or SHA1.

<u>Using Salt Data To Generate Multiple, Unique Index Functions</u>

Using the standard MD5 hash algorithm as an example for the basic index function q(), hashing the high value bit sequence “S” results in the following hash value:

Description	Bit Sequence
S (high-value)	00011011 00001111 11110000 11011000
MD5 Hash*	7d7714e4b82169a38213bda47bdad77a

**MD5 of the string value, not binary – “I said NO SALT in the margaritas...”*

Using a salt value, S can be transformed prior to hashing:

Description	Bit Sequence
S (high-value)	00011011 00001111 11110000 11011000
Salt B=10101010	10101010 10101010 10101010 10101010
XOR (S + B)	10110001 10100101 01011010 01110010
MD5 Hash*	b58de1440afb7b6d8378ad74efef06ed

Multiple salt values can be associated with a given index function:

Function	Salt	Pre-hash Bit Sequence (XOR S ± B)	Label
S (high-value)		00011011 00001111 11110000 11011000	n/a
Index q( )	None	00011011 00001111 11110000 11011000	Q
Index r( )	10101010	10110001 10100101 01011010 01110010	R
Index m( )	10010101	10001110 10011010 01100101 01001101	M

In this regard, for a given salt length L, the number of unique index functions that can be created is 2^L, where B=00000000 would result in the index function q( ) above (no transformation prior to hashing)

Encoding S

In this theoretical example, the “high-value” bit stream (our confidential data) is S:

S = 00011011 00001111 11110000 11011000

F is a random bit string, in this case:

F = 01011100 11110000 10001001 11100001

The random bit string can be generated using a pseudorandom number generator function, or can be digital measurements obtained from a random source (e.g. instantaneous CPU utilization, heat measurements, and the like)

Given F, Q02 can be obtained using q():

Q02 = q(F)

Z and Q03 can be obtained as follows:

$Z = S \pm F$

Q03 = q(Z)

At this point, S can be expressed as:

S = -q(Q02) + -q(Q03)

To reconstruct S, there must be knowledge of the index function q() and the relationships:

Q02 = q(F)
Q03 = q(Z)

As well as the function that defines the relationship between F and Z (which is XOR)

The index function q() can be seeded with many false values of Qn where the associated bit string is not used to reconstruct S, or perhaps Qn is part of another high-value string, T.

Again, from a practical standpoint, q() can be populated with random or pseudorandom bit strings, and the associated labels can be generated using the legitimate hash function h(). This would prevent an attacker from “weeding out” false bit strings by calculating Qn for each Bn, and comparing against q(Bn).

Finally, a blueprint of S can be constructed using the tokens that identify the components used to re-create S:

S = “q,Q02,XOR,q,Q03”

In this example:

q	Identifies the index function -q() used to convert Q02 in to a bitstream
Q02	The label of the first factor. (sequence is inconsequential)
XOR	The logic function used to convert to bit strings back in to S
q	Identifies the index function -q() used to convert **<u>Q03</u>** in to a bitstream
Q03	The label of the second factor.

Note that since Q02 and Q03 are not sequence dependent, they could be listed in any order, meaning Q02 could either be related to F or Z, and the same for Q03. The random and derived factors can be listed in the blueprint in any order.

Q02 and Q03 could conceivably use two different index functions (possibly from two different servers, as an example)

Other transformation functions such as EQU could be used in place of XOR. Unary operations can be added to further convolute bit strings by using bit shifting, inversion, or other “well known” transformations.

Block Encoding

Although the entire bit string S can be encoded as one block, the value of doing this diminishes for large bit strings, since the amount of data in the index function q() is equal to 2 times the length of S, and also increases the chances that compromising q() could result in S subsequently being compromised.

A better approach is to break S down in to fixed-length blocks, resulting in smaller corresponding blocks for F and Z, that are easier to store and transfer, and results in more table entries for q(). Since the sequence of a particular block Qn can't be determined within the bit string S, a brute force attack would not only have to compare all values of q(), but all possible sequences of all values of q() in order to obtain S, which assumes S was known in advance, or else you would not have an end-state to compare against.

Using the example listed previously, and an arbitrary block length of 8 bits, a sample encoding process looks like this:

1. Start with the “high-value” bit string, S

   00011011 00001111 11110000 11011000

2. Break S in to 8-bit blocks:

Blocks of S	Bit String
Block 1 of S (SB1)	00011011
Block 2 of S (SB2)	00001111
Block 3 of S (SB3)	11110000
Block 4 of S (SB4)	11011000

3. Generate random data, F, for each block of S:

Blocks of F	Bit String
Block 1 of F (FB1)	01011100
Block 2 of F (FB2)	11110000
Block 3 of F (FB3)	10001001
Block 4 of F (FB4)	11100001

4. Calculate Z using XOR function from S and F:

Blocks of Z	Bit String
Block 1 of Z (ZB1)	01000111
Block 2 of Z (ZB2)	11111111
Block 3 of Z (ZB3)	01111001
Block 4 of Z (ZB4)	00111001

5. Generate labels and populate the index function, q():

Label q()	Bit String -q()	Purpose
Q06	01011100	Block 1 of F (FB1)
Q07	11110000	Block 2 of F (FB2)
Q08	10001001	Block 3 of F (FB3)
Q09	11100001	Block 4 of F (FB4)
Q11	01000111	Block 1 of Z (ZB1)
Q12	11111111	Block 2 of Z (ZB2)
Q13	01111001	Block 3 of Z (ZB3)
Q14	00111001	Block 4 of Z (ZB4)
Q16	01100001	Random Filler
Q17	01001110	Random Filler
Q18	11011101	Random Filler
Q19	01001111	Random Filler
Q20	00000110	Random Filler
Q21	00011000	Random Filler
Q22	10001110	Random Filler
Q23	01010110	Random Filler
Q24	10000011	Random Filler
Q25	11011010	Random Filler

6. Generate blueprint for each block of S:

Block	Blueprint for Block
SB1	q,Q06,XOR,q,Q11
SB2	q,Q07,XOR,q,Q12
SB3	q,Q08,XOR,q,Q13
SB4	q,Q09,XOR,q,Q14

7. Generate combined “final” blueprint for S by concatenating block-level blueprints

S = q,Q06,XOR,q,Q11;q,Q07,XOR,q,Q12;q,Q08,XOR,q,Q13;q,Q09,XOR,q,Q14

The final “blueprint” string has nothing to do with the original bit string, S

In a practical implementation, the block labels would be non-deterministic, and “real” index rows would not be grouped together – they would be randomly-interspersed with the junk, random data. In this simple example, the block labels are linear and the “real” index rows are grouped together.

Additionally, the block length would have to be significantly larger than the example, in order to make the blueprint as efficient as possible. In a practical implementation, the hash function could be MD5 (cryptographic security is not required) or SHA1, and the block size could be 2k (2048 bytes) or larger. MD5 checksums are 16 bytes long, so the “blueprint” string for each block of S would be 32 bytes, plus the encoding for the index function q(), the transformation function XOR, and possibly location hint data, for a total of 50 bytes or less per block.

In theory, the index function q() would be tied to a specific repository location (such as a URL), and could be defined early in the blueprint string. “Index hint” data might differentiate between multiple repositories on the same server, or different classes of data blocks. For example, random block data could be stored within a file, such as an image on a website, and the “location hint” might be a byte or block offset within the file.

Alternately, various repositories for various index functions could be stored centrally in a directory... knowing which index function relates to which repository, and the location (or multiple locations) for a given repository could be stored in the directory, then referenced within the blueprint string.

Blueprint Mutation / Permutation

A given bit block might be indexed by multiple index functions in order to allow blueprint mutations. For example, a server that hosts two index functions, q() and r(), and has every bit block indexed by both q() and r(), might have a repository that looks like this:

Label q( )	Label r( )	Bit String -q( ) , -r( )	Purpose
Q06	R99	01011100	Block 1 of F (FB1)
Q07	R98	11110000	Block 2 of F (FB2)
Q08	R97	10001001	Block 3 of F (FB3)
Q09	R96	11100001	Block 4 of F (FB4)
Q11	R95	01000111	Block 1 of Z (ZB1)
Q12	R94	11111111	Block 2 of Z (ZB2)
Q13	R93	01111001	Block 3 of Z (ZB3)
Q14	R92	00111001	Block 4 of Z (ZB4)
Q16	R91	01100001	Random Filler
Q17	R90	01001110	Random Filler
Q18	R89	11011101	Random Filler
Q19	R88	01001111	Random Filler
Q20	R87	00000110	Random Filler
Q21	R86	00011000	Random Filler
Q22	R85	10001110	Random Filler
Q23	R84	01010110	Random Filler
Q24	R83	10000011	Random Filler
Q25	R82	11011010	Random Filler

A server might offer a translate function:

http://somesite/TranslateIndex.php?q=Q01&f=r
(might return “R99”, the equivalent of r(-q(Q06))   )

Given the two index functions q() and r() listed above, the following bit strings are all equivalent:

-q(Q06) -q(Q07) -q(Q08) -q(Q09)

-r(R99) -r(R98) -r(R97) -r(R96)

-q(Q06) -r(R98) -q(Q08) -r(R96)

(etc...)

In this example, there are 4 blocks and 2 index functions, yielding 2^4 = 16 possible unique blueprints.

This allows an individual who shares the blueprint to “mutate” the blueprint by randomly translating block labels from -q() to -r() in order to obfuscate the blueprint. Using just two index functions, with an example bitmap consisting of 2,000 blocks, the number of unique blueprints that represent the same bitmap would be 2^2000, which would easily defy signature-based detection at network gateways and when performing server scans.

Advanced server functionality might even offer a “mutate” function, that stores a “meta blueprint” and always returns a unique, mutated version of the original for each download request.

Storage and Transfer of Encoded S

Transferring the encoded high-value bit string "S" depends on neutral third-party servers to host the index function q(), which includes the bit strings and associated labels for all blocks necessary to reassemble S.

The bit strings and labels become "public" information, while the blueprint itself (and perhaps a meta-map of how to interpret the blueprint) would be transferred from peer to peer.

From an implementation standpoint, a simple server script can be written, that performs a database lookup based on a single parameter ("q"):

http://someserver/qindex.php?q=Q08

In our example, this index URL with the q parameter of Q08 would return the byte "10001001" from our simple, sample lookup table.

Other functions might point to mirror sites containing the same index function, or to a global directory URL where additional repositories (index functions) can be located.

Ideally, each factor of S (blocks of F and Z) would be stored on separate servers, and optimally, many different servers.

q1(x) = http://server1/qindex.php?q=x

q2(x) = http://someotherserver2/qindex.php?q=x

Server 1 database:

Label q1()	Bit String -q1()	Purpose
Q06	01011100	Block 1 of F (FB1)
Q07	11110000	Block 2 of F (FB2)
Q08	10001001	Block 3 of F (FB3)
Q09	11100001	Block 4 of F (FB4)
Q21	00011000	Random Filler
Q22	10001110	Random Filler
Q23	01010110	Random Filler
Q24	10000011	Random Filler
Q25	11011010	Random Filler

Server 2 database:

Label q()	Bit String -q()	Purpose
Q11	01000111	Block 1 of Z (ZB1)
Q12	11111111	Block 2 of Z (ZB2)
Q13	01111001	Block 3 of Z (ZB3)
Q14	00111001	Block 4 of Z (ZB4)
Q16	01100001	Random Filler
Q17	01001110	Random Filler
Q18	11011101	Random Filler
Q19	01001111	Random Filler
Q20	00000110	Random Filler

In order for this scheme to work, each index function has to have a globally-unique identifier. By definition, a server/site that hosts a given function q() must also host its inverse function -q().

In a practical implementation, blocks of Z and F would be intermixed, so that a given server might contain blocks of both Z and F, but never the same block number "n" for both, where the server would never contain both ZBn and FBn. The purpose of intermixing blocks of Z and F on the same server is to complicate brute-force analysis – combining blocks of Z together would result in junk data (as would combining blocks of F together), while on any given server, some of the blocks would be missing altogether.

Impact of Block Size

Assuming that a specific file is 4 MB, and the blueprint size per block is 42 bytes (16 per MD5 hash, two hashes, plus 10 bytes of overhead), the following is an analysis of how block size impacts the blueprint file:

Bits per Block	Block Size	Total Blocks	Total Bytes
4,096	512	8,192	344,064
2,048	256	16,384	688,128
1,024	128	32,768	1,376,256
512	64	65,536	2,752,512
256	32	131,072	5,505,024
128	16	262,144	11,010,048
64	8	524,288	22,020,096
32	4	1,048,576	44,040,192
16	2	2,097,152	88,080,384
8	1	4,194,304	176,160,768

As the block size decreases, the blueprint size increases. At 512 bits per block (64 bytes) or above, the resulting blueprint is smaller than the original 4 MB file. At 256 bits per block (32 bytes), the two files are almost the same length. At 128 bits (16 bytes) or less, the resulting blueprint is much larger than the original file.

Larger blocks will ultimately result in fewer hash collisions, and less data stored in the blueprint.

MD5, no longer considered cryptographically-secure, could result in hash collisions – different bit sequences that result in the same hash, and therefore would result in the wrong lookup for the index function, q().

This can be avoided by adding a simple CRC32 (4 bytes per hash) to the blueprint, increasing the block size within the blueprint to 50 bytes, which will virtually guarantee a unique label.

If the blueprint for a given file is longer than the file itself, the argument could be made that the blueprint itself consists of confidential data – defeating the approach of removing information from the blueprint in order to bypass the handling requirements.

Receiving and Reassembling S

The client would obtain the blueprint file for "S", which is a string describing repositories, index functions, transformations, and block sequences used to reassemble the original bit stream of "S".

Once the blueprint file is obtained, reassembly consists of the following steps:

1. Locate the repositories for all index functions. Perhaps this is accomplished via a directory server.

   Example:

   URL for q1 = http://directoryServer/HasFunction.php?f=q1

   returns

   q1(x) = http://server1/qindex.php?q=x

   And

   URL for q2 = http://directoryServer/HasFunction.php?f=q2

   returns

   q2(x) = http://someotherserver2/qindex.php?q=x

2. For each block defined in the blueprint for "S", perform the following steps:
   a) Use the specified index function (repository 1) to download block 1 (B1)
   b) Use the specified index function (repository 2) to download block 2 (B2)
   c) Use the specified transformation to create SBn from B1 and B2
   d) Map SBn in to the bit stream S, based on location "n" (specified in the blueprint)

3. Process the reassembled bit stream "S" (for example, decode the bitstream as a file, and present it to the user)

Legal Analysis

Copyrighted S

The “high-value” bit string S contains confidential data.

Depending on the nature of the confidential data, disclosing the content might impact either the sender or the recipient, or simply *having a copy* of the data might create liability for the sender, recipient, or any hosting facility.

For example, assuming the data is relatively benign, if the sender transmits a confidential list of clients (for valid business purposes) to a vendor or employee, disclosing that list might put the sender at risk for breach of contract (as an example), or knowing that a particular employee has the client list might open them up to subsequent social engineering attacks, where obtaining the list is the target.

Conversely, the data itself might have inherent privacy and security requirements that creates some level of legal responsibility around simply *having* a copy of the data. An example is Protected Healthcare Information (PHI) as defined by the Health Insurance Privacy and Accountability Act (HIPAA). If a patient's medical records are hosted on a server, the owner of the server is required to conform to HIPAA requirements for privacy and security. If the server gets hacked and the information is disclosed, the server's owner could be at legal risk.

Another example is copyrighted material, the disclosure of which could result in impact to revenue, or the illegal copying of which could result in civil and legal penalties.

From a compliance standpoint, data is either in scope or out of scope for privacy and security requirements.

Since the random bit string F is just that – random – it can't be proven to be part of S or a component of S. The specific bit sequence corresponding to F itself could be considered in scope, but a new bit sequence F2 can be generated at random, requiring that all bit sequences be in scope in order to prevent transferring or transmitting F, which is of course not possible.

F is completely safe, and can be any “reference” bit stream. F could even be derived from images taken from a website (making a mirror copy of a website image would, of course violate copyright law, but downloading them dynamically wouldn't)

Z is derived from S and F:

$$Z = S \pm F$$

Assuming S is not known, and F is truly random, Z is therefore random. Because Z is random, the same principle can be applied to Z that we applied to F – if Z was in scope, a new F could be generated, F2, resulting in a new Z2, and this could be iterated infinitely, for each random value of F.

If the complete strings Z and F were stored on one server, with the transformation algorithm well-known, an argument could be made that Z and F together are in scope. By storing Z and F on separate servers, broken in to blocks whose location within the original string S is not known, no case can be made that a given block of Z (BZn) nor F (BFn) could possibly contain S or a part of S, and therefore can't be in scope.

Plausible Deniability

Plausible deniability allows the sender, hosting entity, or recipient to claim no knowledge of the “high risk” string, S.

The most obvious situation where plausible deniability could come in to play, is transferring copyrighted material. For example, if the sender wants to share a copy of a copyrighted song, encoded as an MP3 file, he could encode the file using this method, store the blocks individually on several public servers, and send his friends the blueprint file.

From a legal standpoint, he has not copied nor transmitted the file. The file isn't being hosted on any of the public servers (only encoded bit blocks are being stored there), nor have his friends downloaded the file – they only have the blueprint, which contains no data.

Conversely, having the blueprint file, plus all of the encoded bit blocks on the same system leaves little room for denying that the owner of the system is maintaining a copy of the copyrighted song.

Another situation where plausible deniability would come in to play is corporate or other espionage. In this situation, the sender is in a semi-secure target facility, and must assume all of his transmissions are being monitored. He uploads some seemingly random pictures, text, and other information to various websites, returns “home”, and uses the blueprint to reconstruct the data he smuggled out of the target facility via the 3rd-party websites.

Because plausible deniability implies deception, it seems to infer nefarious intent. This document side-steps any issues surrounding ethics or intent.

Countermeasures

The following techniques can be used to further obfuscate the original source string S, and therefore produce more "random" output blocks, or enhance plausible deniability.

- Many "scans" for high-risk content depend on keyword searches. So for example, a file named "CompanyContactInfo.xls.blueprint", might give the attacker a starting point. Naming the blueprint files with generic names will avoid detection.

- A given server/site might host multiple index functions. Rather than provide a directory of index functions that could provide an attacker with new information, the most secure approach would be to offer an API function that returns a boolean TRUE or FALSE when the client requests a given index function, by unique label. So a website hosting functions q() and r() but not m() might respond as follows:

  http://somesite/HasFunction.php?f=q
  (returns "TRUE")

  http://somesite/HasFunction.php?f=r
  (also returns "TRUE")

  http://somesite/HasFunction.php?f=m
  (returns "FALSE")

- A blueprint is basically a bit sequence, which can be "double encoded" as a second blueprint, in order to shorten the blueprint of S, or further obfuscate S. Additionally, salt data can be stored / encoded as a bit block.

  Extending this approach, S could be deconstructed in to two complete bitstreams that have to be completely reconstructed, and then XOR'd together in order to reform the original "S".

- As mentioned previously, blocks of "Z" and "F" should never be stored on the same server. For a given block n, a server could contain Zn or Fn, but for plausible deniability, hosting some blocks of Z and other blocks of F, should an attacker be able to relate Z, F and S, implies that the hosting server has some knowledge of S. Storing as many random blocks as possible, and re-using some blocks of F, spread across as many servers as possible makes it impossible to prove the intent of the hosting entity, with respect to knowledge of S.

- The index function q() might use a standard hash function such as MD5. Assuming the use of salt data, there is a basis to defy cryptanalysis, because the returned data wouldn't necessarily match the hashed value (D is "Data", L is "Label"):

  D = -q(L)
  -q(L) returns "D".

However,
$L \neq MD5(D)$
because of the salt data included in q()

- If hash values are returned as labels, for example:

  $q(x) = MD5(x \pm S)$ (Where S is Salt data)

  Returned labels should be additionally scrambled using a simple, reversible scrambling function t() (D is “Data”, L is “Label”):

  $L = q(x) = t(MD5(x \pm S))$

  $D = -q(x) = -t(L)$

  t() can be as simple as rotating bytes within the label in some deterministic way – for example, swap byte 1 with byte 7, 5 with 3, etc...

- Non-aligned blocks. Instead of starting at a specific byte offset within the source string, the transformation operation can be specified to start at a byte offset past the block-aligned byte offset of 0

- Overlapping transformations. A buffer string can be used as an intermediary, where additional transformations can be applied.

- Many diverse transformations and multiple streams. This could include bit splitting, where odd and even bits are split in to diverse bit streams.

  $B1 = S \times 01010101$
  $B2 = S \times 10101010$

  $S = B1 + B2$

  (Note: x means “AND”, + means “OR”)

- Inversion. Unary operations such as inversion against one block.

- Reversible operations, such as XOR against an arbitrary constant value (salt).

- Storing blocks on many diverse servers. The more spread out the data, the less data per individual repository, making cryptanalysis much more difficult, and deniability much more plausible.

- Use well-known bit sequences that are subject to copyright or in the public domain. These could be images from a website, or some other arbitrary source in place of F. The resulting Z bit string is more deterministic, but could still be suitably obfuscated by other countermeasures. This could be enhanced by embedding meta data in to image files, or via true “steganography”, where hidden data is embedded in the image pixels.

- Alter the content. For example, an attacker might look for a specific digital artifact within the original content, as proof that a file was copied, or originated from a specific source. Transcoding content (audio, video, text) to another format would erase any digital artifacts.

  Additionally, if some quality can be sacrificed, consider converting to analog and back to digital. For example, a confidential document could be printed, then re-scanned in order to erase any digital watermarks that could tie the original file back to a specific person.

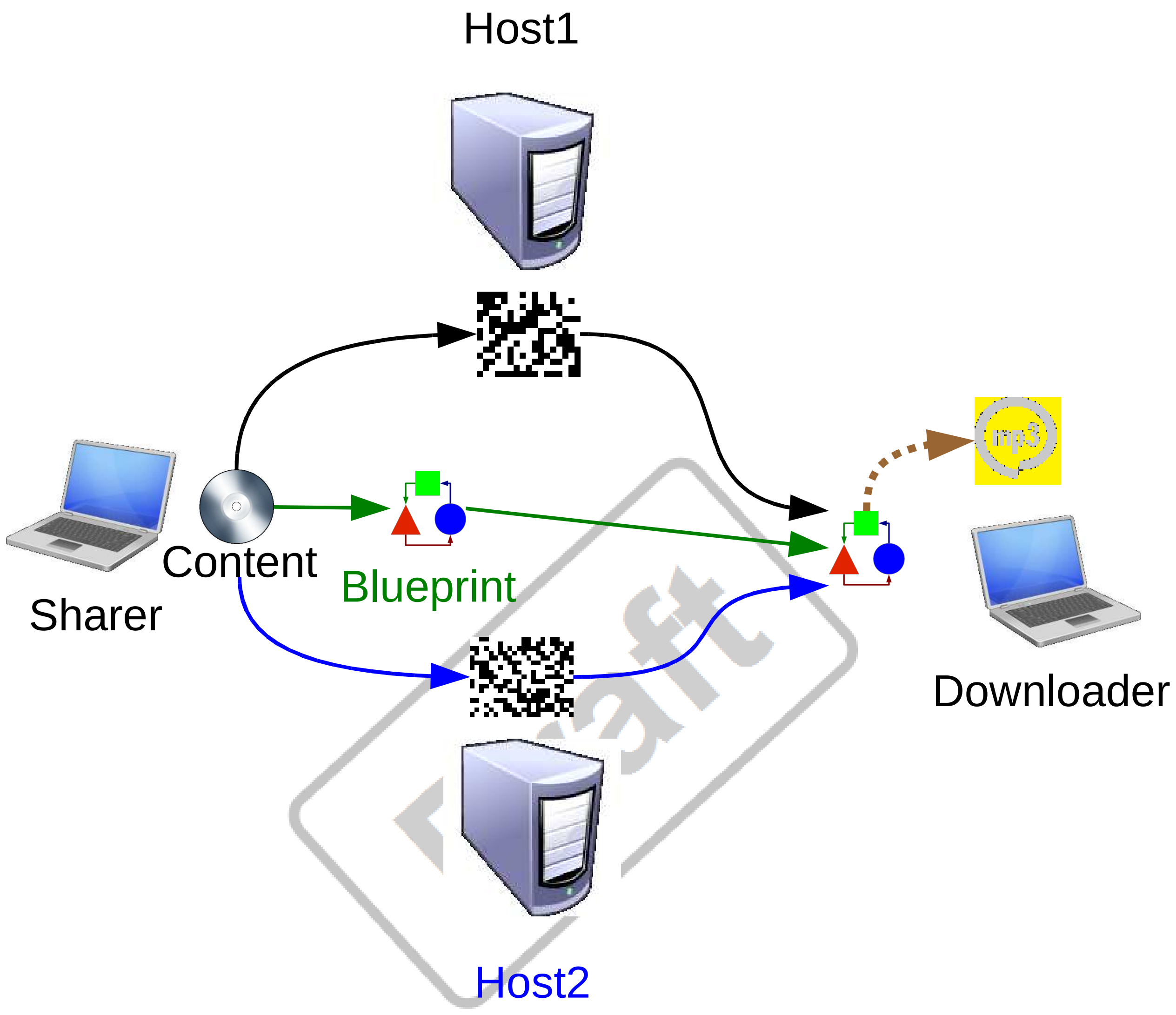
- The blueprint doesn't contain any data. Likewise the individual bit blocks don't contain any data. For the best plausible deniability, and least liability, bit blocks should be downloaded on the fly, only stored in RAM, and never stored on physical media. Likewise, the “final” bit stream should be dynamically reconstructed, and never stored on physical media. Due to advanced computing capabilities, this can be done as effectively as streaming.

- An attacker might configure a “honeypot” server, pretending to host labeled bit blocks for a given function (maybe “q()”). The system should employ a mechanism by which the client can validate that the label in question refers to the block that was downloaded. Without compromising the underlying hash algorithm, nor the index function itself, this is difficult to do. The easiest approach is to include a hash checksum with the data itself, such that the hash is also encrypted alongside the data. For example, the first x bytes of every data block (once decoded) could be the checksum.

Analysis of File Sharing Schemes, Compared to the Blueprint Method

The “blueprint” method, described in this document provides an alternative to various, existing file transfer / sharing methods. This section compares these to the blueprint method, using a “worst case” scenario of sharing copyrighted content. The example used is sharing a copyrighted song stored in MP3 format, and was selected as the example because it presents significant liability for multiple parties.

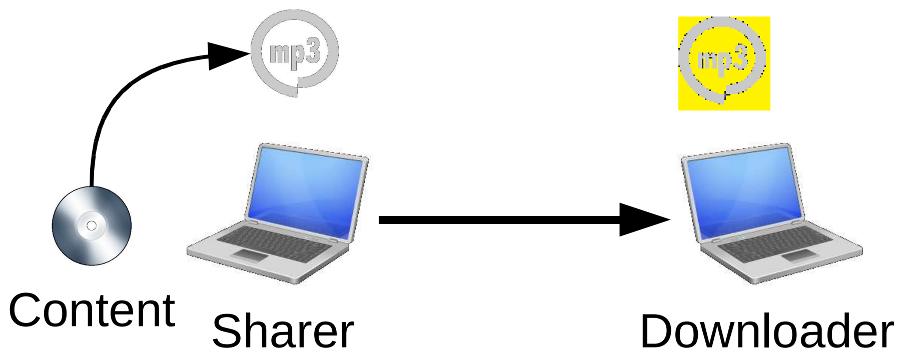
Overview of Blueprint Method



Using the blueprint method, the content is split apart in to multiple streams, broken in to blocks, labeled, uploaded to various public servers (Host1 and Host2), and the labels are stored in the blueprint. The Sharer either directly or indirectly transfers the blueprint to the Downloader.

The Downloader only stores the Blueprint, downloading the bit blocks on the fly, reconstructing the content only a segment at a time, in RAM.

Direct Sharing



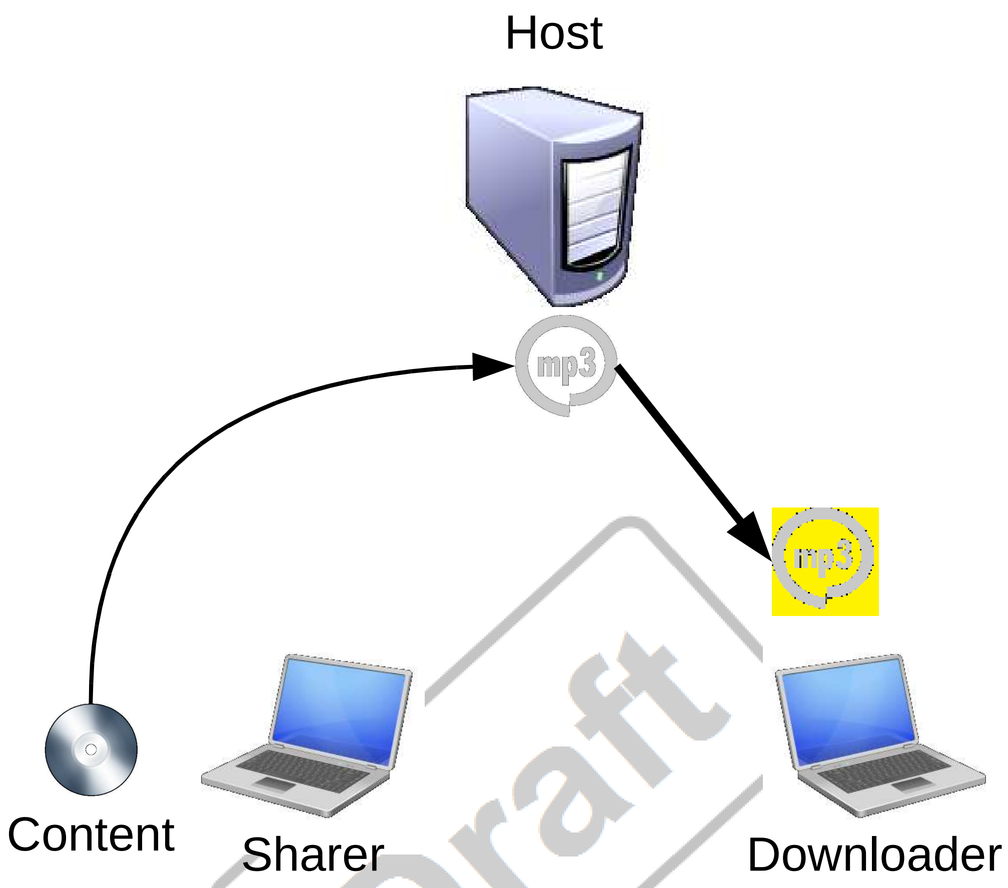
The sharer transcodes the content to a digital file, which is then transferred to the downloader.

The Sharer is legally liable for sharing copyrighted content, while the Downloader possesses an illegal copy, and is also liable.

Analysis:
Since the blueprint contains no content, the sharer is not liable. Likewise, the Downloader is not liable for storing a local copy of the blueprint. Should the Downloader use the blueprint to reconstruct and store a local copy of the original file, he/she would be liable for the illegal copy of the content.

If the Downloader pulls blocks from multiple sources, while downloading bogus blocks, reconstructing the content on-the-fly, it would be impossible to prove that the Downloader made a copy of the content. This would be logically-equivalent to the public liability of viewing a Youtube video.

Hosted Sharing



The Sharer transcodes content that is stored on a host server. The Downloader downloads the content from the host server.

The Sharer has no liability, as long as the content isn't tagged. Tagging ties a specific copy of the content to a specific distribution, meaning that based on serial number or some other tag embedded in the content, it can be tied back to the Sharer. If that's the case, two opposing arguments can be made:

- The Sharer is knowingly making the content available to be downloaded, and maintains liability

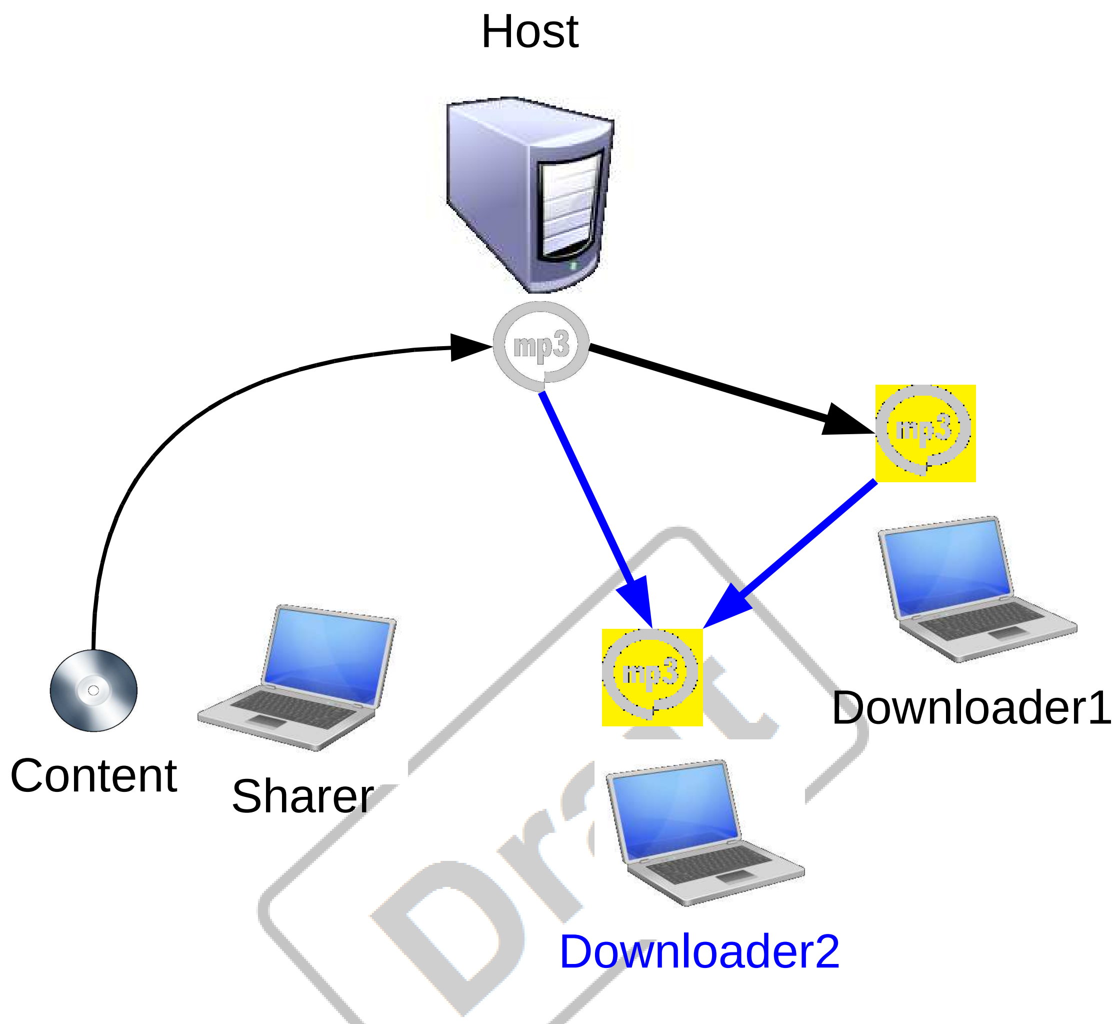
- The Sharer has the right, per Digital Millenium Copyright Act (DMCA) to make a backup copy of the content, and isn't responsible for security of the Host.

The Host is 100% liable for storing and serving copyrighted content – this is the exact situation faced by YouTube and other video sharing sites, where they have no direct control over what their members upload.

The Downloader is liable only to the extent that they “illegally” posses copyrighted content.

Analysis:
As with direct sharing, there is no liability in hosting the blueprint.

Peer to Peer (P2P) Sharing



The Sharer transcodes and creates a "seed" copy of the content, stored on the Host. The first Downloader downloads content from the Host, while the second downloader downloads content from both the Host and the first Downloader, who has now become a Host.

Originally, P2P sharing required a centralized "directory" of content stored on various hosts – each Downloader was also a Host by default.

Newer P2P sharing methods don't use a central directory. Instead, "seed" locations and block information are stored in a "torrent" file (similar to a blueprint), which decentralizes the entire process. If one "seed" host goes offline, the file can be re-seeded from another host.

The ephemeral nature of P2P sharing is believed to be a protection from detection and liability. From an enforcement standpoint, all the attacker has to do is attempt to download the file using a standard torrent client. If the file name contains "SomeBand - SomeSong.torrent", then the expectation is that the download and reassembly process will result in a file containing content that is copyrighted by "SomeBand". Downloading the file and reviewing the content would validate for the attacker whether or not the Host is actually sharing copyrighted content.

Typically, enforcement centers around the most prolific Hosts, but any person who downloads the content, and then makes it available for subsequent downloading is potentially, significantly liable.

P2P depends on a large network of mostly unreliable hosts, precluding streaming, and necessitating a complete download of the file before being able to use the content.

Like the blueprint method, there is no direct liability for hosting a torrent file – it contains no actual data.

Analysis:
In contrast, the blueprint method allows content to be streamed from a smaller network of more reliable hosts using well-known locations for a particular bit block, while presenting very little liability for the hosting servers. A blueprint contains no data, and can be shared, hosted, or downloaded with no liability.

Naming the blueprint "SomeBand – SomeSong.blueprint" will obviously draw unwanted attention. Unlike the torrent method, an attacker who reassembles the content can't prove that any given Host has a complete (or even partial) copy of the content. For example, a blueprint could be devised that would allow "SomeBand – SomeSong.blueprint" to be completely reconstructed using bit blocks found exclusively within images stored on a web site – the images themselves are subject to copyright, independent of the file being reconstructed. The attacker him/herself would be downloading copyrighted content, using it in violation of its owners' copyrights, in order to prove whether the content of "SomeBand – SomeSong.blueprint" actually contains the content in question.

Conclusion

By attacking the concept that content is something we store locally, we can construct a method by which content is dynamically reconstructed using specific blocks of random data.

This approach provides a framework for the plausibility of openly sharing and hosting random data that can be securely reassembled in to the original high-value / high-risk content.