2017

Cryptography and Secret Sharing Theory in Game Design.

KRISTIAN F DAVIDSEN

Introduction

Cryptography games are not a new concept. Newspapers have had them for decades under the cryptogram form. Most cryptography games however lack substance, they either present the same puzzle with minor variations, or they are a lightning round style brain game. These are often intended for play by those with knowledge or interest in cryptography.

This proposition will aim to utilize cryptography and secret sharing theory to construct a game for those without an interest in cryptography. It should be recognized though that people must have base interest in the underlying construct and aesthetic of a game for them to enjoy it. For cryptography, this would be most likely puzzles and logical gameplay with the aesthetic of challenge, man vs. machine. It is possible to include competition, man vs. man, but this proposition will focus on building a single player cryptography based game for those who may be interested in puzzle solving, but varying amounts of interest in cryptography.

Ciphers in games

Methods, Goals, and Benefits

There are a few different ways to implement cryptography in games depending on what the end goal of the game is. For instance, if you were creating a game for a cryptography class the goal would ultimately be to teach. In this case the game would do little to none of the calculations. It would find interesting and engaging ways to show and teach the algorithms and necessary operations. The fun that would be derived from this would come from the solving of the math problem. The other major way to implement cryptography would focus purely on its use to create puzzles. The game would perform any required algorithms freeing up the player to focus on solving the overarching puzzle. The fun here is in the logic, like a mystery novel. The game design being proposed here work to achieve a middle ground for the overall game changing the individual style of each problem to suit its complexity.

Constructing a Cipher Problem

This is a proposed system for designing a cipher based problem.

First it must be decided what is the ultimate objective. What is the player attempting to? This is universal for game problem designing. For cipher problems, this will almost always be either finding the cipher text or the plain text. There may be cases where the solved problem will then be used elsewhere, but that is considered a different problem on its own with its own objective and settings.

Second is deciding what information the player has available to them. In this model, this will be split into the primary and secondary clues. The primary clues are given to the player at

the onset of the problem. For a cipher example, this would often be either the cipher or plain text, whichever was not chosen as the objective, something like an encoded note or password. This clue should be obvious. The player should know what they are looking at and what they need to do with it. Without this tool solving the problem without help should be near impossible. The secondary clues are smaller things, the cipher key for example. This category would also include hints to the player on how to use the tools they've acquired. These clues can be somewhat optional. Finding them helps, but they may not be entirely required. For ciphers, these clues include things such as variables in the algorithm, directions on how to do something, etc.

Finally comes the function. This is how the player solves the problem given the tools and directions they have available to them. The work done here by the player comprises most of the puzzle. This is plugging the variables into the problem and following the directions. This is where the player feels like a player and accomplishes something. Implementing a good function into a narrative often requires the designer to be creative. Turning numerical answers into coordinates, alphabetic answers into names or messages, it's the designer's job to engage the player in an interesting and meaningful way as opposed to simply having a lightning round problem game with a point score.

Considering the problem design model and the methods of and goals of implementation, there are three primary ways to create a problem. The first is Augmented Reality (AR), being asked by the game to use outside sources to solve a problem. This puts a lot of the burden on the player, and because of that isn't used too often. A common mainstream example is looking up a line number in a play to get a clue for a puzzle. For a cryptography game, this would likely include things such as looking up how to do a shift cipher, perhaps presented in an interesting way. It would be a bad idea to use this type of problem for a more mathematically heavy puzzle. It is unreasonable to expect the average player to want or be able to learn the RSA algorithm to progress. Asking such would cause most player to leave the game and not bother. The second method of implementation is the "Plug and play" method. In this case the machine does the heavy mathematical lifting. This removes burden from the player and opens up the logic portion of the puzzle. This style of problem works better with a narrative supporting it, and is more useful for the complex algorithms such as RSA and El Gamal. Finally, taking a middle ground of the two results in a designer directed problem. The player is given hints that may or may not be the actual mathematical formula, but allows them to solve it none the less. These are given out by the game through NPCs (Non-Player Characters) and the like. This design would still use the machine to do the hard work while still giving the player some cipher solving to do. This particular game design will use the complexity of the problem to determine which problem creation method is best. High complexity problems work best with the Plug method, low complexity problems with the AR, and medium complexity problems with the designer directed method.

Implementation

Low complexity alphabetic ciphers such as the Caesar Shift Cipher are some of the easiest to implement into a game because they already are games to some extent. Cryptograms are an alphabetic cipher that is already used for newspaper puzzles. Let's create an AR for a simple Caesar shift cipher.

For this problem, the objective will be the original text, and the starting clue will be the cipher text:

E Q Y C T F U F K G O Z P A V K O G U D G H Q T G V J G K T F G C V J U

The secondary clue will be: Shakespeare, Julius Caesar 2.~~2.33~~ → with the crossed-out section being completely unreadable. The point of this clue is to point the player in the direction of the shift cipher. To find the function, it is important that the player has been introduced to the concept of AR puzzles in the game, whether this is the first one or not, and that they know this is one. It shouldn't be too far of a stretch to imagine that a player trying to solve this would google "Caesar code" after a few times which yields the shift cipher. The number and arrow then give the final parts of the formula, the step number and direction. Using those clues the plain text turns out to be: "Cowards die many times before their deaths" which is the 33rd line in Julius Caesar, Act 2, Scene 2. You could also avoid the AR by putting the information into the game via a library or NPCs for instance.

A type of problem that would fit into the directed problem would be a simple binary stream cipher. Stream ciphers use randomly generated streams of characters to construct a key. For this problem, we will let the plain text be the goal, though regardless the process is the same. The primary clue here will actually be the key. The key is given on a computer that requires a password to access certain parts, and would be randomly generated. The key has to be XORed with the plain text which would have to be found by the player. This would be the secondary clue. Finally, the function for this problem would be XORing which could be taught to the player. An easy way to do this is through examples. By presenting completed problems to the player, they could figure out that combining 0 and 0 or 1 and 1 results in 0, and that 0 and 1 or 1 and 0 results in 1. Let us put this into a numerical example. Accessing the computer gives this key:

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

The player finds can then find examples of complete problems on the computer in an obvious folder labeled solutions:

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

\+ 1 0 0 1 1 0 1 0 0 1 1 0 1

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

Given a few of these the player should reasonably be able to figure out what the operation is. They then find the cipher text which is found somewhere in the room:

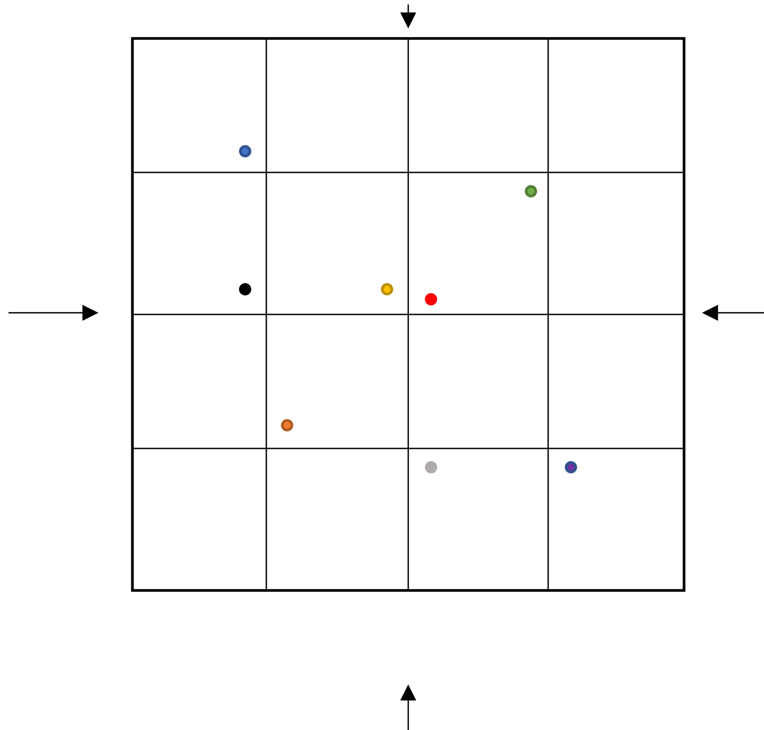
0 1 1 0 1 0 1 0 1 0 1 0 0 1

The computer then asks for an input which is the plain text:

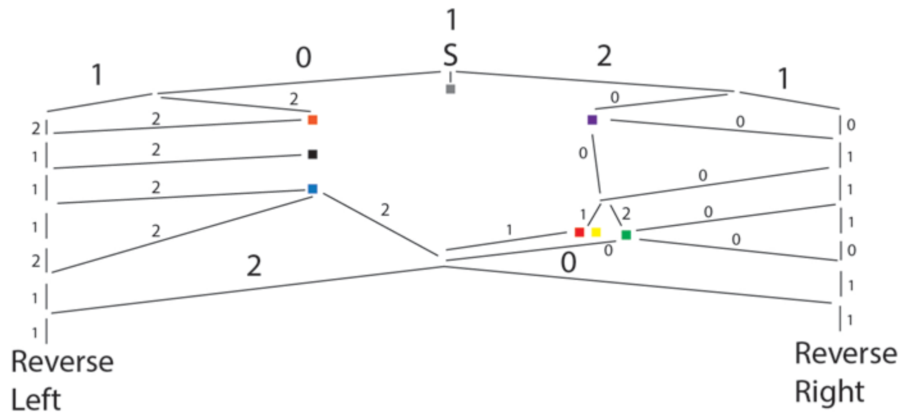
```
  1 1 0 0 1 0 1 0 1 0 1 0 1 1
+ 0 1 1 0 1 0 1 0 1 0 1 0 0 1
  1 0 1 0 0 0 0 0 0 0 0 0 1 0
```

These strings shouldn't be too long so that the player can remember or write them down with no trouble. These binary strings can also be interpreted as any binary decision such as yes or no, left or right, etc. The operation and answer are simple enough that a lot can be done with it.

Another similar cipher that can be implemented is the prefix cipher. Prefix ciphers are ciphers constructed using a decision tree, commonly binary. Starting at the top, the user follows a sequence until they reach a letter. The letter is then encoded as the sequence that For this implementation, the goal would be the completed tree and we will use a form of code based off the Huffman code, though instead of letters, it will be based on the frequency of locations. The main clue would be a map of a city grid with different locations.



The goal here is to find the most frequent location from the four entry points. At each fork there are three directions: left, right, and forward. This results in tertiary trees based on the frequency from and entrance. Let left = 0, right = 2, and center = 1, starting from the south results in a tree that looks like so:



It can be found that from the southern starting point; the most frequent destination is grey, followed by purple. By creating prefix trees for the remaining starting points, the player can guess which location is the most frequent. It’s important to note that the player can also brute force this problem and ignore the cryptography element allowing those who don’t wish to take the time deducing the correct answer to still be able to continue. Similar to the stream cipher implementation, the prefix code can also be adapted to any problem that requires few, continuous choices such as directions and yes or no questions.

We will demonstrate the Plug type of problem through a commonly taught cipher, the RSA cipher. RSA codes utilize the fact that prime factorization for large numbers is very difficult. For a plug problem, the player’s satisfaction comes from the finding of the clues and use of the solution. The actual solving of the problem is done by the computer as it is too complex to expect a player to learn for a game. For this problem, the goal will be the plaintext which will be a 4-digit code, an address perhaps. The player already knows the cipher text. The remaining values used for decoding RSA are hidden and must be found by the player. Specifically, they must find n and d. The mathematics of an entire problem, including the work done by the computer is this:

Let p and q be generated prime numbers

$$p \equiv 71, q \equiv 59$$

Let n be the multiplication of p and q

$$n \equiv p * q \equiv 4189 \quad e \equiv 3$$

Let phi be the totient of p and q

$$\varphi(n) \equiv (p-1) * (q-1) \equiv 4060$$

While e is normally published, in this particular case the player has no use for it, unless the designer wishes to add more things for the player to find and enter into the computer to get the answer.

$$Let\ Plaintext \equiv x \equiv 1234$$

Let y be the cipher text

$$y \equiv x^e \ mod\ n \equiv \ 1234^3\ mod\ 4189 \equiv 229\ mod\ 4189$$

y is a given clue. The player is told that 229 is the number they need to convert by entering it into the computer. How easy it is for them to know that 4189 is another necessary component is up to the designer to decide.

Let d be the modular inverse of e which is needed to decode the cipher text

$$d \equiv \ e^{-1}\ mod\ \varphi \equiv 3^{-1}\ mod\ 4060 \ \equiv 2707\ mod\ 4060$$

2707 is another part that the player needs to find x. Like n, how easy or hard it is to determine its use depends on the problem and designer.

$$x \equiv y^d \equiv 229^{2707} mod\ 4189 \equiv 1234$$

A look at how the computer handles its part of the problem can be seen with this code. Credit goes to Rod Stephens from csharphelper.com for his random prime generation code and to Samuel Allan and Jeff Lambert from stackoverflow.com for their modular inverse code. The following pseudocode is for C#, the Unity code language. The full code is located in the appendix.

Choose p = prime number between 100,000 and 999,999
Choose q = prime number between 100,000 and 999,999
Multiply p * q = n
Multiply (p-1) * (q-1) = phi
Let e = 17
Find Modular inverse (e, phi) = d
Let the plaintext = 1032
Calculate ciphertext as Modular Power (plaintext ^ e mod n)
Recalculate plaintext as Modular Power (ciphertext ^ d mod n) to prove answer
Display results p, q, n, e, d, ciphertext, and plaintext

The remaining code consists of the functions used for finding modular inverses and probabilistically choosing prime numbers in a range. This code is also included in the appendix. This code demonstrates the probabilistic way to generate primes, which isn't that useful to games as it can be hard to line up specific randomly generated content to the right seeds. Running the code provides this example:

p = 704,857
q = 684,769
n = 482,664,223,033
e = 17
d = 28,391,931,377
Cipher Text is 52,258,105,436
Plain Text is 1032

Since the plain text matches, the code worked.

Finally, another type of cipher that's implemented through the plug method is the Diffie–Hellman key exchange. While this isn't a cipher, it utilizes many of the same ideas and allows for a more interesting exchange of data. For Diffie–Hellman the player represents one side while and NPC or computer plays the other. The goal in this case for the player is to get the shared key. This key could be used later for a different problem. Here is an example key exchange with player as A and the NPC as B:

Let p be a prime number and g be a primitive root of p

$$p = 103$$
$$g = 70$$

Let a be a number chosen by or given to the player and A be the player's secret

$$a = 34, A \equiv g^a mod\ p \equiv 70^{34} mod\ 103\ \equiv 56$$

The player computes their part of the exchange, A, to give to NPC in return for their part.

Let b be a number generated by the computer for the NPC and B be the NPC's secret

$$b = 86, B \equiv g^b mod\ p \equiv 70^{86} mod\ 103\ \equiv 97$$

With the NPC's part, the player is then able to compute the secret S.

$$S_1 \equiv B^a mod\ p\ \equiv 97^{34} mod\ 103\ \equiv 46$$

To make sure it is correct, the game computes both.

$$S_2 \equiv A^b mod\ p \ \equiv \ 56^{86} mod\ 103 \ \equiv 46$$

The next page contains a pseudocode example of the Diffie–Hellman key exchange and uses a different method to generate the primes and find the roots that is more useful for game design and even used for crypto security. The full code is included in the appendix.

Create an array consisting of 20 primes from 101 to 197
Create an array consisting of 20 corresponding primitive roots. The first entry is a primitive root of 101, the second of 103, and so on
Initialize and choose a random number between 0 and the length of the arrays
Let p = the element from the prime array determined by the random number
Let g = the element from the primitive root array determined by the random number
Let a = 34
Calculate Modular Power (g ^ a mod p) = A
Let b = 86
Calculate Modular Power (g ^ b mod p) = B
Calculate Modular Power (B ^ a mod p) = s
Calculate Modular Power (A ^ b mod p) = d to check against secret key s
Display results p, g, a, b, A, B, s, and d

This example is also in C# and uses the array method for selecting primes. Instead of the computer probabilistically finding primes and having a potential to be wrong, a list of pre-generated primes is fed to it instead. The program then selects a random element from this list as well as a corresponding primitive root to use for the problem. This guarantees that the prime numbers are in fact prime numbers and makes it easier to match them with one of their primitive roots using the same method. These arrays are much larger when used in security, but for games a smaller list is easier for the designer to control. Running this code gives the example:

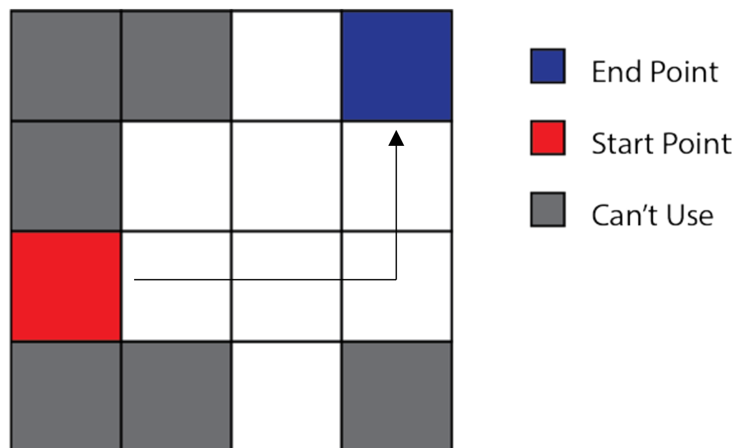
p = 137
g = 47
a = 34
b = 86
A = 37
B = 9
s = 136
d = 136
Common Secret is 136

Once again it is important to check one answer against the other to make sure the code worked.

The next section of this design proposal will deal with secret sharing theory in game design.

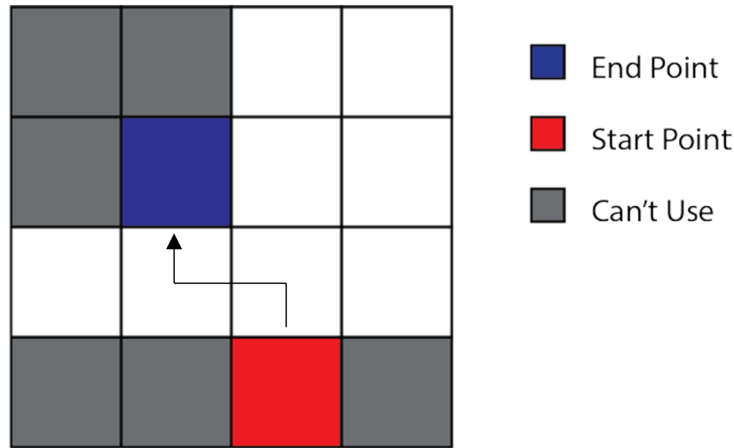
Secret Sharing in Games

The use of secret sharing theory in games is more prevalent than one might think. Secret sharing theory at its core is finding an answer that completes certain requirements. For example, in the Shamir scheme, the answer is the y-intercept of a polynomial and the requirements are the shares showing where the polynomial lies. In the classic bank vault problem, the answer is any combination of employees that can access the vault, and the requirements are the clearance levels for each employee. This concept is used extensively throughout games, both digital and analogue. Almost any puzzle found in a game is just a single player secret sharing scheme. Old adventure games such as The Legend of Zelda with their block puzzles have a solution where the block must be in a certain spot with the requirements being that it must travel along certain paths. Even this iconic puzzle has the components of a secret sharing scheme. The secret is the way to get the block from the starting point to the end goal, while the access structure is the set of specific movements to reach the end.



While I doubt game designers made these puzzles with secret sharing in mind, it is certain that secret sharing and game design are related. In these cases the designer does not care if the scheme is secure or ideal in the context of cryptography because the purpose is to create a single player challenge. And while a game system can be cryptographically secure because the player acts as all the players in the scheme, I believe that a game designer's definition of a secure scheme is that the game is fair, i.e., that a player with a set of information is as close to finding the answer as a player with different, yet equal amounts of information. The classic scheme of intersecting lines on a 2-D plane with the (x, y) coordinate as the secret is neither secure nor ideal in cryptography as the individual players have more information than someone with no shares. In a single player version of this however using the designer's definition, a player with two intersecting lines is as close to the answer as another player with two different intersecting lines that intersect at the same location. In the block sliding puzzle for example, the

scheme is secure, or “fair”, if after changing the block’s starting location, the solution can still be reached. In this example the puzzle is still completable.



Talking about security in game designed secret sharing systems leads to a discussion on ideal schemes. For the first ideal requirement, that a system must be secure, we will use the game designer’s definition that secure equals fair. For the second requirement, I think it is acceptable to use the pre-existing definition, that the range for the secret is the same as the range for the shares. Going back to the block example, the secret is a square on a 2-D grid that the block must reach, and the shares are a sequence of squares that the block travels, so this scheme is ideal. Note that a scheme does not have to be ideal to make a good game, but a game should be fair to be good.

A good game to look at secure and ideal secret sharing schemes is the party game Bomb Squad. In this game the players are each given pieces of instructions on how to defuse a bomb. This is obviously a secret sharing scheme as we have individual players with their own shares and a secret that requires multiple shares to find. First, is the game cryptographically secure? No because each share gives some insight in how to defuse the bomb as opposed to no information forcing random chance. This assumes that all information in one player’s instructions is correct and there are no red herrings. If this assumption is false, then a player’s individual share is untrustworthy and thereby useless on its own, making the game cryptographically secure. Is the game secure by game definition or fair? This first requires defining fairness in a cooperative setting. Fairness for multiplayer games is often defined by the community as equal opportunity from the start; each player has the same opportunity to contribute to the outcome as the others. By this definition, Bomb Squad is sometimes fair. If the information each player receives is as useful as the other players, it is, otherwise, it is not. It also matters if we are looking at the fairness of multiple rounds or each single round. Assuming that the game is in fact fair, then is it ideal? The answer once again is “it depends”. The correct set of instructions is a set length, so if the player’s information has filler to make it the same length as the correct answer, the game is ideal.

Secret sharing schemes can also be used to promote fairness in competitive games. Fog of war, the inability to see areas of a map often found in Real Time Strategy games (RTS), follows secret sharing theory. For this discussion we will ignore the idea of spectators, people who have the combined information of all the players. The shares the players have are what they are able to see, while the secret is all information that is on the map. This scheme is not cryptographically secure because some information in these games is always better than no information. Each player has partially revealed the secret. Following the equal opportunity definition for multiplayer fairness, the system is fair. Fog of War behaves the same for all players. If one player can get closer to revealing the whole map in a particular way, so can the other players. Is this system ideal though? No, it is not because the shares are always smaller than the secret for any realistic scenario. If a player has revealed the whole map without cheating, the game is for all intents and purposes decided.

Problem Creation

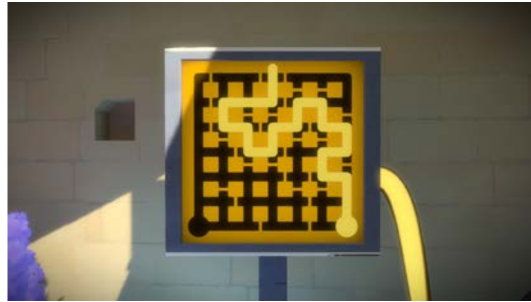
Creating a detailed process in implementing secret sharing theory into game problems is difficult due to the openness the theory presents. As mentioned previously, secret sharing theory can be abstract, like fog of war, or very basic, like Bomb Corp. It can even be mathematically heavier like CRT or Shamir's scheme. The most important part with designing these problems is figuring out what the scheme is out to accomplish. Is it being used as a puzzle, act as a game mechanic, or something else? If it is being used as a puzzle, what is the ultimate goal? Another very important part of implementing schemes is for the player to know what their goal is. It is alright for them to need to figure it out on their own, but it shouldn't be too ambiguous. This proposal will include a small variety of different types of secret sharing schemes focused on the single player and puzzle use of the theory. These schemes will be neither secure not ideal.

Implementation

A simple game implementation is the classic intersecting lines scheme. The goal is to find the treasure and the hints are the lines. The player goes about collecting the lines from clue given by the game. In this way they start with either one or no shares and it is the goal of the player to get the rest. They then use the shares to figure out where the item, person, etc. they are looking for is.

Another simple implementation is the circuitry, or connecting paths problem. In this problem, the shares are different edges and the secret is getting from one specific vertex to another. The access structure are the sets of edges that allow for this connection. A common occurrence of this scheme in games is creating flow. Connecting two electrical components to create current and turn on power and connecting water pipes using specified paths have been

common in games for a while. A recent puzzle game, The Witness, uses these schemes extensively.



http://cdn.gamer-network.net/2016/usgamer/p1_07.jpg

Another common, and old, puzzle example is riddles. The shares and access structure for riddles are more complex, but they still follow secret sharing theory. Take the classic riddle:

“What walks on four legs in the morning, two at noontime, and three in the evening?”

The shares in this case are the pieces of the riddle and any outside knowledge the person attempting to solve it may have. The secret is a correct answer and the access structure is a set of words and descriptions that are solutions, human, man, women, people, etc.

Number based riddles are also possible. Let’s say that the player is attempting to figure out how many apples are available to buy at a store. They learn by asking around that the store was shipped apples in packs of five. It then repackaged them into sets of three for selling, but one apple was bad, so one pack only has two. They also learn that there are an odd number of apples and by looking there can’t be more than fifteen of them. This is actually a Chinese Remainder Theorem secret sharing scheme. The different pieces of information compose the shares and the solution is the secret. This can be solved using the CRT method, more importantly; it can also be solved naively. By simply going through the numbers one through fifteen and figuring out which numbers meet the set of conditions, the player can find the answer to be eleven.

As problems get more complex, the narrative around them has to grow as well in order to accommodate the player. It is important that the player does not always require knowledge of the algorithm in order to solve the problem. Let’s create a CRT problem backwards, starting with the math and numbers, and then trying to design a good problem around it.

CRT:

We star by defining our conditions for x

$$x \equiv 5 \; mod \; 11$$

$$x \equiv 12 \; mod \; 17$$

$$x \equiv 19\ mod\ 83$$

$$x \equiv 36\ mod\ 59$$

Then we find the answer modulus N

$$N = 11 * 17 * 83 * 59 = 915{,}739$$

Solve for each condition's y value

$$y_1 = \frac{915{,}736}{11} = 83{,}249$$

$$y_2 = \frac{915{,}736}{17} = 53{,}867$$

$$y_3 = \frac{915{,}736}{83} = 11{,}033$$

$$y_4 = \frac{915{,}736}{59} = 15{,}521$$

Solve for z which is the modular inverse of y mod m, m is the original condition

$$z_1 \equiv 83{,}249^{-1}\ mod\ 11 \equiv 1$$

$$z_2 \equiv 53{,}867^{-1}\ mod\ 17 \equiv 14$$

$$z_3 \equiv 11{,}033^{-1}\ mod\ 83 \equiv 69$$

$$z_4 \equiv 15{,}521^{-1}\ mod\ 59 \equiv 15$$

Then we calculate the answer

$$x \equiv (5 * 83{,}249 * 1) + (12 * 53{,}867 * 14) + (19 * 11{,}033 * 69) + (36 * 15{,}521 * 15)\ mod\ 915{,}739 \equiv 260639\ mod\ 915{,}739$$

This is a more complex CRT problem than the one from before. So now we'll design a problem around it. We start with what the player is trying to figure out. Let's make it the final number x = 260,639. So, what is "x"? Let "x" in this case is a password to a file on a computer. This computer belongs to a serial killer and the player is a detective trying to get into the file. The killer is overconfident and likes to leave clues in the room for the detective to find. The clues could range from a morbid "19 down out of 83" to silly "the store only had five of the eleven oranges I wanted". Since this CRT problem isn't as trivial as the previous one, it would likely require the use of an in-game calculator. This could be disguised as "HiNt.exe" on the computer and would have entry fields that looked like "_____ out of _____". Inputting the clues found around the room would result in a related output. Depending on how many steps you want the player to go through, this could be the y values that then require more programs

and clues, or it could skip straight to the end. Either way the challenge and enjoyment in this problem comes from the discovery aspect of the clues rather than figuring out the secret on one's own. This is the same type of problem as the previous one, but simply changing the complexity results in a much different experience.

Let's take a look at another more numeric heavy secret sharing scheme that will require little to no in-game calculator use, Shamir. Shamir's scheme uses points to construct a polynomial. The shares are the y coordinates of the points, and the secret is the y-intercept. The scheme is a (n, k) threshold scheme where there are n shares and you need k of them to find the secret. Similar to the CRT, we will construct the entire problem first, and then build a narrative around it. For this particular example, we will use small numbers which will become apparent when we discuss a possible narrative.

Shamir:

Start by deciding what the secret S is

$$S = 2$$

Then we define the (n,k) threshold which for this problem is

$$n = 4, k = 4$$

Then we decide on the shares

$$y_1 = 4, y_2 = 2, y_3 = 2, y_4 = 10$$

So the points are $(1,4)$ $(2,2)$ $(3,2)$ $(4,10)$

Using polynomial interpolation we get:

$$l_1 = \frac{-2}{3}x^3 + 6x^2 - \frac{52}{3}x + 16$$

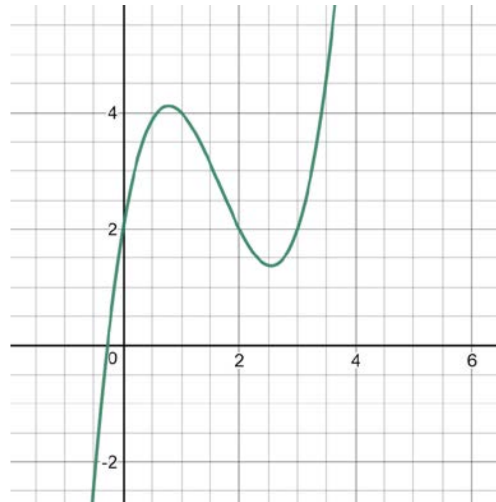
$$l_2 = x^3 - 8x^2 + 19x - 12$$

$$l_3 = -x^3 + 7x^2 - 14x + 8$$

$$l_4 = \frac{5}{3}x^3 - 10x^2 + \frac{55}{3}x - 10$$

Which gives the complete polynomial $f(x) = 2 + 6x - 5x^2 + x^3$

Since Shamir's scheme returns a polynomial, the best way to implement it is to view it as a geographical figure. The plot for this particular polynomial is:



Constructing a narrative for a geographical feature is much easier than trying to make one for a numerical figure. Lets have it so that the polynomial represents a physical river, and the player is attempting to find a burried treasure. The tresure's location has yet to be defined to be on the river. The player then goes around trying to find the individual shares, the y coordinates, and puts them into some machine or function that they found, and it gives the respective x coordinate. As the player finds more points, they start to deduce that the points are all on the river. They then find some other clue, something that says "follow the mossy log". This log would lie on the y-axis of the graph and tell the player that the secret is on this axis. This hint combined with learning that the tresure is on the river would lead the player to the y-intercept which is the Shamir Scheme secret. Note that the player is not required to know Shamir's scheme in order to find the secret. To help make things more interesting and prevent the player from finding the secret too easily, the polynomial formation could not show up on the map, or the final clue pointing towards the y-axis could be gated behind finding the other points first. This also works for n values greater than k. The player only needs to find k out of the n possible pieces of information. Making this work for numerically larger polynomials only requires a scaling of the map to smaller units, they could even be made up units.

In conclusion cryptography can be used for game design and development in a way that does not just present the problem with high scores and timers. Secret sharing has also been used in games for quite some time and is still a useful technique for game design. This proposition has given methods to create game problems from cryptographic and secret sharing theory concepts that appeals to a wider audience than the expected cryptographic game.

Appendix

RSA code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;
namespace System.Security.Cryptography
{
  namespace ConsoleApplication3
  {

    class Program
    {
      static void Main(string[] args)
      {
        BigInteger p = FindPrime(100000, 999999, 10);
        BigInteger q = FindPrime(100000, 999999, 10);
        BigInteger n = p * q;
        BigInteger phi = (p - 1) * (q - 1);
        BigInteger e = 17;
        BigInteger d = modInverse(e, phi);
        BigInteger plaintext = 1032;
        BigInteger ciphertext = BigInteger.ModPow(plaintext, e, n);
        BigInteger ptsolve = BigInteger.ModPow(ciphertext, d, n);
        Console.WriteLine("p=" + p);
        Console.WriteLine("q=" + q);
        Console.WriteLine("n=" + n);
        Console.WriteLine("e=" + e);
        Console.WriteLine("d=" + d);
        Console.WriteLine("Ciphertext is " + ciphertext);
        Console.WriteLine("Plain Text is " + ptsolve);
        Console.ReadLine();
      }

//Modular Inverse
static BigInteger modInverse(BigInteger a, BigInteger n)
      {
        BigInteger i = n, v = 0, d = 1;
        while (a > 0)
        {
          BigInteger t = i / a, x = a;
          a = i % x;
          i = x;
          x = d;
          d = v - t * x;
          v = x;
        }
        v %= n;
        if (v < 0) v = (v + n) % n;
        return v;
      }
      //Prime Generation
      static BigInteger FindPrime(int min, int max, BigInteger num_tests)
      {

        // Try random numbers until we find a prime.
        for (;;)
        {
```

```
            // Pick a random odd u.
            int u = Rand.Next(min, max + 1);
            if (u % 2 == 0) continue;

            // See if it's prime.
            if (IsProbablyPrime(u, num_tests)) return u;
          }
        }
        static Random Rand = new Random();
        static bool IsProbablyPrime(int u, BigInteger num_tests)
        {
          checked
          {
            // Perform the tests.
            for (BigInteger i = 0; i < num_tests; i++)
            {
              // Pick a number n in the range (1, u).
              BigInteger n = Rand.Next(2, u);

              // Calculate n ^ (p - 1).
              BigInteger result = n;
              for (BigInteger power = 1; power < u - 1; power++)
              {
                result = (result * n) % u;
              }

              // If the final result is not 1, p is not prime.
              if (result != 1) return false;
            }
          }

          // If we survived all the tests, p is probably prime.
          return true;
        }

      }
    }
}
```

Diffie–Hellman

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;
namespace System.Security.Cryptography
{
  namespace ConsoleApplication4
  {

    class Program
    {
      static void Main(string[] args)
      {
        int[] primes = new int[] {101,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,197 };
        int[] roots = new int[] {29, 70, 38, 6, 54, 110, 50, 47, 88, 32, 77, 66, 122, 91, 45, 11, 63, 145, 52, 5};
        Random random = new Random();
        int primep = random.Next(0, primes.Length);


        BigInteger p = primes[primep];
```

```
            BigInteger g = roots[primep];
            BigInteger a = 34;
            BigInteger A = BigInteger.ModPow(g, a, p);
            BigInteger b = 86;
            BigInteger B = BigInteger.ModPow(g, b, p);
            BigInteger s = BigInteger.ModPow(B, a, p);
            BigInteger d = BigInteger.ModPow(A, b, p);
            Console.WriteLine("p=" + p);
            Console.WriteLine("g=" + g);
            Console.WriteLine("a=" + a);
            Console.WriteLine("b=" + b);
            Console.WriteLine("A=" + A);
            Console.WriteLine("B=" + B);
            Console.WriteLine("s=" + s);
            Console.WriteLine("d=" + d);
            Console.WriteLine("Common Secret is " + s);
            Console.ReadLine();
         }
      }
   }
}
```

Works Cited

Miyamoto, Shigeru. "The Legend of Zelda."

Blow, Jonathan. "The Witness."

Bogost, Ian. How to do things with videogames. University of Minnesota Press, 2011.

Bower, Craig P. Unsolved! The History and Mystery of the World's Greatest Ciphers from Ancient Egypt to Online Secret Societies. Princeton University Press, 2017.

McGonigal, Jane. Reality is broken: why games make us better and how they can change the world. Penguin Press, 2011.