UNIVERSITY OF CALGARY

Towards code obfuscation through video game crowd sourcing

by

Sutapa Dey

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

December, 2014

Abstract

Code obfuscation is used to prevent software piracy as well as to avoid reverse engineering of malicious software (malware). From the perspective of malware authors, unique approaches to obfuscate code are beneficial to avoid detection. There have been instances where malware authors have leveraged the support of unsuspecting humans for malicious activities. In this work, we analyze if humans can obfuscate code without knowing that they are generating code. We opted to examine if video games could be a possible channel to extract code obfuscations from humans.

In this thesis, we discuss the development of a new game that is designed to generate obfuscated code during gameplay. We also research if code obfuscations are possible using existing games. We assess our implementation to check if it is feasible to generate diverse obfuscated versions of original code based on the randomness generated in the game due to player interactions or movements in video games. Lastly, we discuss the limitations of this approach and recommend some future work.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. John Aycock for giving me enormous support and advice in my Master's study and research. It has been a really great experience working under his supervision.

I would like to thank Michael Locasto and Beaumie Kim for serving on my thesis committee.

I thank my fellow labmates in the Double-Secret Security Lab/Programming Languages Lab: Sarah Laing, Daniel Medeiros de Castro, Jonathan Gallagher and Subashis Chakraborty for their support and critical feedbacks. I would also like to thank my friends Rashmi Kumari, Tannistha Maiti, Enaiyat Ghani Ovi, Shamim, Prashanth Kumar, Madeena Deena Sultana and Padma Palash Paul for their immense patience and understanding during the last couple of months before my defense.

Special thanks to my mom, Purabi, for giving me continuous support and for inspiring me to pass through hardships in my graduate life. And another special thanks to my fiancé, Siddhartha Datta, for his patience and support during my study.

Lastly, I would like to thank the supreme force for supporting me spiritually throughout my life.

Table of Contents

List of Figures and Illustrations

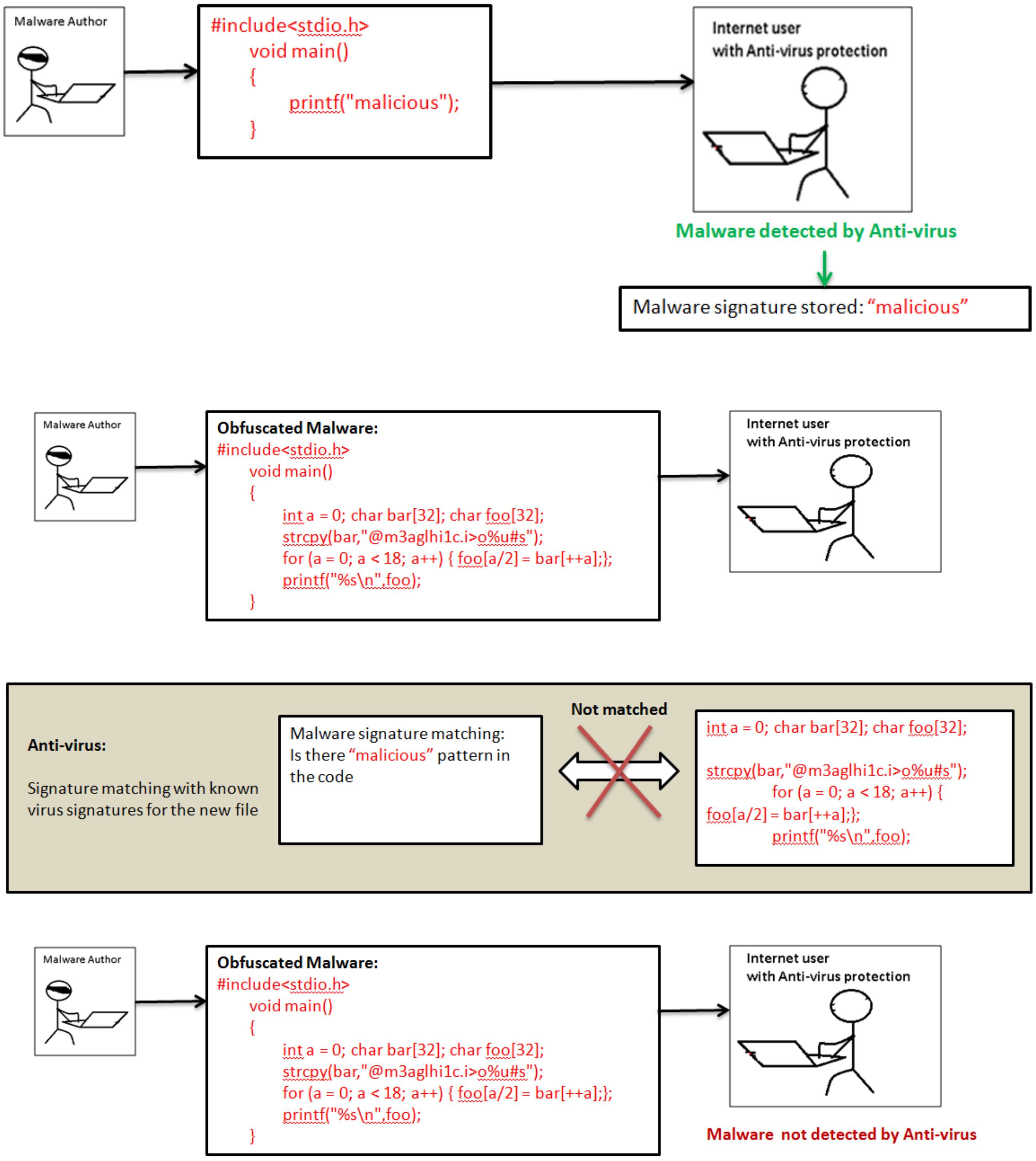
Chapter 1

Introduction

This research work is a method to achieve code obfuscation. Code obfuscation is one of the technical solutions for software intellectual property protection. As explained in this paper [1], it is the process of transforming software code such that it becomes difficult to understand for a human adversary or for automated tools trying to reverse engineer the code. There are several ways to obtain obfuscating transformations. Removing comments or inserting ambiguous comments in the code. Changing the layout of the program by assigning variables with confusing or non-meaningful names. Transforming the program semantics by inserting bogus conditions in loop statements and adding unnecessary operands to math expressions.

The technique of code obfuscation has been used to discourage software piracy. It is also used by malware (malicious software) writers to make it difficult to detect or reverse engineer malicious code [6]. Anti-virus software used a mechanism known as signature-based detection to detect malware. A signature was often based on a sequence of instructions within the malicious code. As discussed in this paper [6], malware using code obfuscation can change the sequence and hence avoid detection by anti-virus programs (Figure 1.1).

Source code obfuscation could be done with the help of obfuscator tools or manually [1]. Novel techniques to generate obfuscated source code are beneficial to an adversary. This research work analyzes a unique technique to produce obfuscated code using humans playing video games. This type of obfuscation could be achieved either developing a customized game which can obfuscate code or modifying an existing game to extract data for subsequent code obfuscation (Figure 1.2).


Malware Author
Internet user
with Anti-virus protection
Malware detected by Anti-virus
Malware signature stored: "malicious"
Malware Author
Obfuscated Malware:
Internet user
with Anti-virus protection
Anti-virus:
Signature matching with known virus signatures for the new file
Malware signature matching: Is there "malicious" pattern in the code
Not matched
Malware Author
Obfuscated Malware:
Internet user
with Anti-virus protection
Malware not detected by Anti-virus

```
#include<stdio.h>
      void main()
      {
            printf("malicious");
      }
```

```
#include<stdio.h>
     void main()
     {
          int a = 0; char bar[32]; char foo[32];
          strcpy(bar,"@m3aglhi1c.i>o%u#s");
          for (a = 0; a < 18; a++) { foo[a/2] = bar[++a];};
          printf("%s\n",foo);
     }
```

```
int a = 0; char bar[32]; char foo[32];

strcpy(bar,"@m3aglhi1c.i>o%u#s");
          for (a = 0; a < 18; a++) {
foo[a/2] = bar[++a];};
          printf("%s\n",foo);
```

```
#include<stdio.h>
     void main()
     {
          int a = 0; char bar[32]; char foo[32];
          strcpy(bar,"@m3aglhi1c.i>o%u#s");
          for (a = 0; a < 18; a++) { foo[a/2] = bar[++a];};
          printf("%s\n",foo);
     }
```

Figure 1.1: Obfuscated malware avoiding signature based detection

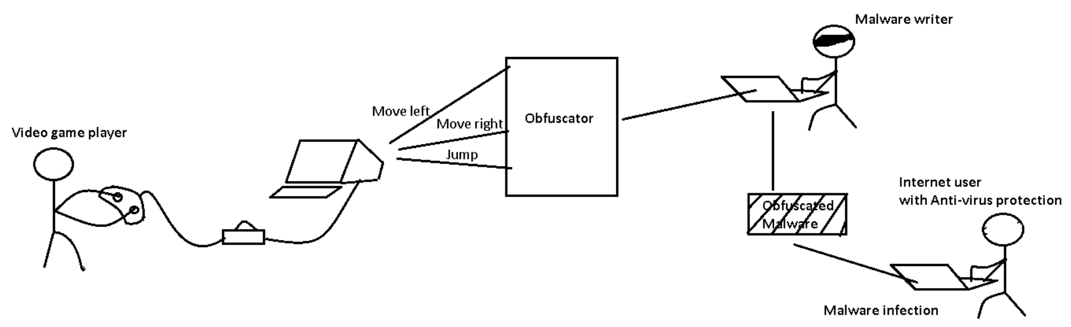


Figure 1.2: Use Case - Video game obfuscating malware

1.1 Objective

Humans have been used for novel purposes such as solving scientific puzzles and thus devising a working solution. One such example is the game known as FoldIt [29]. It is a multi-player game which engages the player in solving puzzles involving protein structures. The game interface is easy to understand and use, constrained by simple rules. Also the player does not need to have in-depth knowledge of biology to participate and play this game. Some other examples include InnoCentive [30] and Duolingo [28].

There have also been instances where unsuspecting humans have been exploited for malicious activities. One example would be that of porn CAPTCHA [34]. Spammers designed a Windows game which made the players solve CAPTCHA images (benefiting the spammers) to access the pornographic content in the game. Humans could be tricked to click on links or install software which can compromise the security of their personal computer systems. Then the compromised systems could be used for malicious activities such as spamming or used as one of the machines in a botnet.

This research work is also an attempt to trick humans into programming obfuscated code while playing an altered version of a video game. The obfuscated code later could be used for malicious purposes.

Why video games? Video games provide the player with excitement based on audio-visual interactions and inherent challenges. Hence it is probable that players would not be suspicious of any malicious activities in the background while they are playing the game. Moreover, gameplay may vary from player to player based on the challenges in the game. Thus it may be possible to spawn diverse obfuscated versions of the same code from different players.

Another perspective could be that games in general engage players in a competitive activity constrained by a set of rules. Similarly, programming languages are defined by a set of rules or syntax which is used to develop instructions for computers. So it may be possible to map the set of rules in the game to the set of rules of a programming language and, consequently, generate code during gameplay based on the mapping.

1.2 Research Design

We investigated two approaches to analyze the feasibility of using humans playing video games as the "obfuscator" for a given piece of source code. The first one was to develop a video game which is customized to produce obfuscated code when played. The second approach was to test if an existing game is also capable of producing obfuscating transforms with suitable alterations.

Our first approach gave us an insight into the game design process and how we can map game elements to code elements in the source code. Also, we saw how operations in the code could be represented by player interactions with non-player characters in the game. It also helped us to analyze sources of randomness in the game which impacts the obfuscated code generation process. That is, different obfuscated code will be generated every time the game level is played. It is probable that the random sources identified based on player character movement could be used across games of different genres.

The second approach was a validation of the findings from our experience from the first approach but with some constraints. We could not map code elements to game elements since we did not design the game. But we were able to use the same sources of randomness from the first game to generate obfuscated code from an existing game.

1.3 Significance

One major contribution of this research is to verify if video games could be exploited for malicious activities. Based on our research, it is probable that humans playing video games could be exploited in helping an adversary with harmful intent. As a next step, we could step up our defense mechanisms to prevent such attempts if the adversary decides to employ this technique.

1.4 Thesis Outline

We present a background review in Chapter 2 of three domains on which our research is based on: code obfuscation, crowd sourcing, and visual programming languages. We discuss examples of obfuscated code and code obfuscation based on compilers and pseudo-random number generators. We review different techniques for crowd sourcing such as microwork as well as malicious uses of this approach.

We discuss the steps of developing a new game which is customized to generate obfuscated code when a player is playing the game level in Chapter 3. The elements in the game represent the code elements. Player interaction with the game elements is mapped to operations in the code. The conflicts or enemies in the game as well as player movement in vertical and horizontal direction are used to generate the obfuscating transforms in the code. This game used different colors to represent code elements. Hence we reviewed color standards to differentiate between similar colors. Our review analysis is also presented in Chapter 3.

As a next step, we analyze and discuss the feasibility of generating obfuscated code using an existing game in Chapter 4. The game selected for this analysis is Super Mario. The game is similar in genre with the game we discussed in Chapter 3. This similarity is preserved while selecting an existing game so that we can apply the same techniques for code obfuscation. Since we did not design the existing game, however, we could not create the mapping between game and code elements. We explained how we overcame this lack of flexibility in the case of Super Mario. We examined various sources of randomness in Super Mario and analyzed them with the NIST randomness test suite. These tests examined if the sources are random enough to be used for obfuscating transforms.

We present limitations of our work in Chapter 5, along with future work which could extend the research idea that we have investigated in this work.

Chapter 2

Related Work

Our work presents a new method of code obfuscation leveraging crowd sourcing techniques using video games. The related work falls into three areas: code obfuscation [1], crowd sourcing techniques [17], and visual programming languages [40].

2.1 Code obfuscation

Code obfuscation is a dual-use technology to modify source code (refer to Listing 2.1) for the purpose of logic obfuscation to protect intellectual property [1] as well as for malicious purposes, to make malicious code harder to detect or analyze. Obfuscation is used to make it difficult to reverse engineer the code to help in, as Collberg mentions in [1, pg 1], "technical protection of software secrets".

There are many techniques used for obfuscating code; this is not an exhaustive list. One of them is to insert dead or irrelevant codes (Listing 2.2) [1]. In Listing 2.2, the first if-block in the method division is dead code[1] as the condition will always be true and it is not used for any computation in the program. Another obfuscating transformation Collberg mentions in his paper [1] is to use lexical transformations such as scrambling identifiers (Listing 2.2). Listing 2.1 has meaningful variable names such as "numerator", "divisor" and "quotient". This helps in understanding the division logic in the code much faster. In comparison, Listing 2.2 has scrambled variable names. Hence reverse engineering of such code will take more time as Collberg states the "pragmatic information" from the variable names has been removed [1, pg 10].

[1]Dead code is a piece of code which gets executed when the program is run but does not contribute to the computation in the program [56].

Redundant operands could be used for obfuscating math expressions using algebraic laws (Listing 2.3) [1]. In Listing 2.3, the value of the expression after the "+" symbol is always zero. Another method of code obfuscation is using dummy program methods to conceal program flow (Listing 2.3) [2]. The method "doSomething" in Listing 2.3 is a dummy method which always returns true.

```
public class DivisionObfuscation {
 public static void division(int numerator, int divisor){
  if(divisor==0){
   System.out.println("Divisor cannot be zero");
   return;
  }
  int quotient = numerator/divisor;
  System.out.println("Result:  "+quotient);
 }
 public static void main(String[] args) {
  division(4,0);
 }
}
```

Listing 2.1: Original Java code

```
public class DivisionObfuscation {
 public static void division(int i0, int i1){
   if(true)
   {
    if(i1==0){
     System.out.println("Divisor cannot be zero");
     return;
    }
   }
   int _001 = i0/i1;
   System.out.println("Result: "+_001);
 }
 public static void main(String[] args) {
  division(4,0);
 }
}
```

Listing 2.2: Dead code insertion and scrambling identifiers

```
public class DivisionObfuscation {
 public static void division(int i0, int i1){
   if(i1==0){
    System.out.println("i1 cannot be zero");
    return;
   }
   int _002 = 1;
   int _00x1 = 30;
   int _001 = i0/i1+_002*30-_00x1;
   System.out.println("Result: "+_001);
 }
 public static void main(String[] args) {
  division(4,2);
 }
}
```

Listing 2.3: Using redundant operands for obfuscation

```
public class DivisionObfuscation {
public static void division(int i0, int i1){
  if(doSomething())
   {
    if(i1==0){
    System.out.println("i1 cannot be zero");
    return;
    }
   }
   int _002 = 1;
   int _00x1 = 30;
   int _001 = i0/i1+_002*30-_00x1;
   System.out.println("Result: "+_001);
  }
 }
 public static boolean doSomething(){
  boolean res = true;
  return res;
 }
 public static void main(String[] args) {
  division(12,12);
 }
}
```

Listing 2.4: Using dummy method for obfuscation

Code obfuscation has also been used to protect malicious code (malware) [6]. Another purpose was to avoid signature-based detection (detecting common malware byte sequences) by virus scanners [7, 8]. The first approach was to encrypt[2] the malware code to hide its signature and include a decryptor loop to decrypt and run the encrypted code. As Peter Ferrie indicates in his paper on X-raying techniques [20], viruses have used substitution ciphers, running keys (different key values for encryption each time) and random keys from pseudo random number generators for encryption. As analyzed by Ferrie [20], it may be possible to recover the decryption key by using simple cryptanalysis techniques as well as by analyzing the decryptor part of the virus body. Hence the encryption schemes used by the viruses are not cryptographically secure. When the infected file is run, the decryptor program will decrypt and execute the malware [7, 8]. Later oligomorphic and polymorphic malware [7, Part 1] used various code obfuscation techniques such as dead code insertion and register reassignment [7, 9, 10] to produce many versions of the decryptor program. In the previously mentioned malware, code obfuscation was being applied to the decryptor loop of the malware. In the next generation of malware known as metamorphic malware [7, 10, 8, 9], the *entire* malware code is obfuscated to produce different variants of a malware program. These obfuscated malware programs have distinct code differences but they produce the same result [6]. This property made it difficult for anti-virus programs to detect metamorphic malware [6]. Designing the metamorphic engine of such malware could be challenging as metamorphic malware needs complex code obfuscation procedures [12].

2.1.1 Code obfuscation based on compiler optimization

An optimizing compiler transforms a given program to its more time-efficient and resource-efficient version [11, pg. 383]. Transformations include using equivalent instructions which are faster, removing redundant operations as in Figure 2.2, using one

[2]In the field of computer viruses and malware, encryption is viewed as a form of obfuscation rather than used as a secure medium of communication as in cryptography.

```
void code()
{
 int res=0;
 for(int i=1;i<10;i++){
 res=res+i;
 }
}
```

```
void code()
{
 int res=0;
 //Arithmetic
    Progression Sum
    Formula
 res=(int) ((9*(9+1))
    /2);
}
```

Figure 2.1: Replacing several operations by one operation

compact operation which can replace several consequent operations as in Figure 2.1 and using algebraic laws to produce simple instructions [11, pg. 383]. These transformations could be applied to a single instruction, a loop-free sequence of instructions, basic blocks, loops, or to the entire program [11, pg. 383]. Superoptimization was introduced by Massalin in her paper [14]. It takes a given machine code as input and searches for alternative code of shortest size as described by Massalin [14] with signum function in Figure 2.3. Massalin observed in [14, pg. 122] and supported by the example in Figure 2.3, "startling programs have been generated, many of them engaging in convoluted bit-fiddling bearing little resemblance to the source programs which defined the functions". Massalin's work on superoptimization was not focused on software protection. But from the example, it is evident that this technique could also be leveraged as a source of code obfuscation. This paper [21] suggests a similar approach by using superoptimization to generate obfuscated code for the purpose of software protection. However the optimal sequence for code instructions will be the same or vary slightly each time optimization is done, so the number of obfuscated or optimized code versions for the same input code will be limited.

```
void code()
{
 int t=0;
 while(t<50*2){
  t++;
  }
}
```

```
void code()
{
 int t=0;
 int check = 50*2;
 while(t<check){
 t++;
 }
}
```

Figure 2.2: Removing redundant operation

```
signum(x)
int x;
{
 if(x>0) return
    1;
 else if(x<0)
    return -1;
 else return 0;
}
```

Listing 2.5: signum(x) in 9 instructions

```
(x in d0)
add.l d0,d0 | add d0 to
   itself
subx.l d1,d1 | subtract (dl+
   carry) from dl
negx.l d0 | put (0-d0-carry)
   into d0
addx.l d1, d1 | add (d1+carry
   ) to d1
```

Listing 2.6: signum(x) using dl (4 instructions)

Figure 2.3: Superoptimization example using signum function (from [14])

2.1.2 Code obfuscation based on pseudo-random number generation

A sequence of random numbers as Donald Knuth describes in his book, *"The Art of Computer Programming"*, is characterized by the following properties - (1) the numbers in the sequence were selected "merely by chance", (2) the numbers in the sequence have no dependency on each other and (3) each number in the sequence could be selected from "any given range of values" (refer to Figure 2.4) [32, Section 3.1]. Knuth mentions several mechanical methods used to generate random numbers such as rolling dice and a random number machine known as "ERNIE" to select the winning numbers in the British Premium Savings Bonds lottery [32, Section 3.1]. Mechanical methods are non-deterministic ways to generate randomness as the sequence is not predictable [33]. Another approach to generate random sequences is to use deterministic mathematical algorithms such as linear congruential sequences [32]. Each number in the sequence is applied as input to the algorithm to produce the next number. Deterministic methods are suitable to meet the required randomness for many applications [33]. The deterministic mathematical algorithms are prone to looping which means after generating a specific sequence of numbers, the sequence is repeated. Hence as Knuth mentions in his book, the deterministic methods generate "apparently random" sequences "often called pseudo-random or quasi-random sequences in the highbrow technical literature". For the rest of the thesis, we will use "random number generator" interchangeably with "pseudo-random number generator" (PRNG).

Recently, one novel idea was proposed to use a pseudo-random number generator (PRNG) for code obfuscation [3]. It may be possible to match the bytes of code with a random number sequence generated by standard PRNGs such as the Mersenne Twister with an appropriate seed. The seed value is hidden and not stored in the obfuscated software. Hence, this implies there is no code to decode or reverse engineer

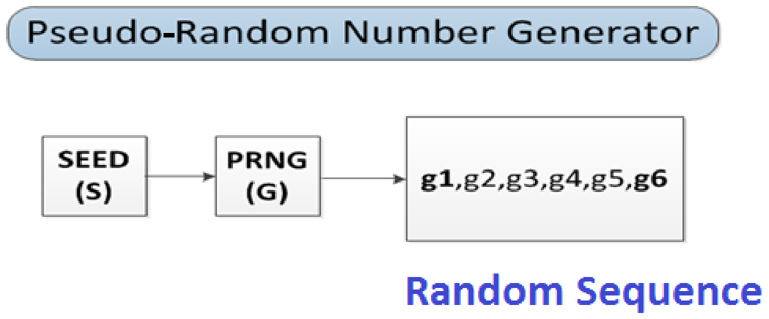


Figure 2.4: Pseudo-random number generator

until and unless the PRNG seed (which can re-generate the bytes of the code) is known. The obfuscated code could update the seed value remotely from a web server like the Conficker virus [4] or could compute it as an environmental key of the host machine [5].

Searching for the relevant seed values requires high computing power to process a large range of seed values and analyze which seed can be used to obfuscate the software code. In case of an adversary trying to obfuscate malicious code, having control of botnets [13] with a million or more computers could substitute for the high computing power [3].

2.2 Crowdsourcing

Crowdsourcing was the term coined by Jeff Howe and Mark Robinson, editors at Wired Magazine, in 2005, while discussing how the Internet is being leveraged by companies to outsource work [22]. In Howe's definition of crowdsourcing as quoted in [22], "The act of taking a job once performed by a designated agent (an employee, freelancer or a separate firm) and outsourcing it to an undefined, generally large group of people through the form of an open call, which usually takes place over the Internet." In short, crowdsourcing involves engaging the crowd especially through on-line methods to solve a problem [17]. Microwork is a crowdsourcing concept, which provides monetary benefit in exchange for solutions to problems which cannot be solved by computers [23, 24]. Amazon's Mechanical Turk [25, 26], reCAPTCHA [27] and Duolingo [28] are a few existing microwork prototypes. Another innovative use of crowdsourcing was to involve people to contribute to scientific research. Some examples include Foldit [29], InnoCentive [30] and linear logic models derived from a game of graphs in mathematics [31].

2.2.1 Microwork - crowdsourcing with benefits

Microwork (Figure 2.5) is a form of crowdsourcing which offers financial rewards for performing outsourced work. There are several applications which leveraged this model to outsource work. Amazon's Mechanical Turk [25, 26] allows "requesters" (businesses or individuals) to post "human intelligence tasks" or HITs on the Amazon website. Workers or "turkers" can search and select relevant tasks and get paid on completion [25]. Calgary, Alberta-based iStockphoto allows amateur as well as professional photographers to sell their photos through their portal and shares a percent of the purchase price [30]. Duolingo uses crowdsourcing to translate books written in different languages to digitize them. Luis von Ahn [28] developed the idea of Duolingo but wanted it to be a free service as well as crowd-sourced. So Duolingo was modeled as a language learning service which served both purposes [28].

2.2.2 Crowdsourcing for scientific purposes

There have been several instances where the on-line community has been engaged to solve crowd-sourced research problems. Foldit [29] is an attempt to use the "human visual problem-solving and strategy development capabilities" to solve a "computationally-limited scientific problem" such as protein prediction challenges. Foldit [29] is an interactive multi-player game where players fold or combine components of a protein into a definite structure. The simple interface and initial training

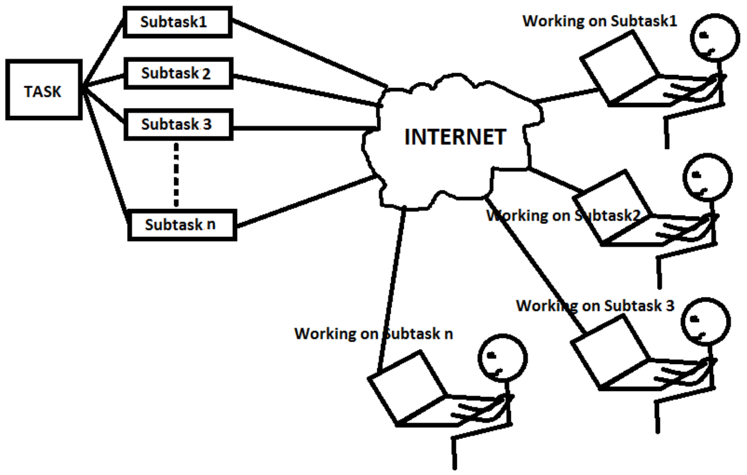


Figure 2.5: Microwork

levels of the game helps in engaging players with no in-depth knowledge of biology or science. InnoCentive provides the opportunity for the global scientific community to earn money while designing innovative solutions for research challenges [30]. The reCAPTCHA is another dual purpose crowd-sourced application to protect against spamming[3] as well as to help in digitizing books [27]. It is the next version of CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) with the added purpose of digitizing books [27]. As mentioned in the paper [27], optical character recognition (OCR) software is used to scan text books to digitize them. Words not recognized by OCR are solved by humans as distorted images in reCAPTCHA.

2.2.3 Has crowdsourcing been used for malicious activities?

As mentioned in the paper [35], CAPTCHA images represent an "unsolved Artificial Intelligence (AI) problem". The paper [35] also prescribes that CAPTCHA offers a secure solution against email worms and spam as it is difficult for automated scripts to decipher CAPTCHA images. Faced with CAPTCHA challenges, spam developers

[3]Sending unwanted bulk emails for the purpose of advertising is known as spamming and the emails are known as spam [36].

used a crowd-sourcing technique to solve the problem as mentioned in this news article [34]. They uploaded pornographic material to the web which could be accessed when a CAPTCHA image is solved. People visiting pornographic websites or whose machine was infected with malicious software were targeted. The solved CAPTCHA image is then used for creating email accounts for spamming.

2.3 Visual programming languages - inception of an idea

The purpose of our game described in the next chapter was to find ways to represent obfuscating transforms in the game. R. Halprin and M. Naor discuss in the paper [39] how humans playing video games could be used as a source of entropy or randomness. In a similar fashion, we tried to leverage human player inputs via video games to generate obfuscated code. The reason to use video games as a medium is to allow contributions from a larger population without necessarily having any prior knowledge of programming. Visual programming languages[4] [40] are a probable technology that can be used to design such games. The graphical elements in the game could be chosen in such a way that the player will be oblivious to the code generation happening while the game is being played.

Visual programming languages provide the option to generate programs while interacting with icons or graphical elements, for example, ToonTalk[43]. As Ken Kahn explains in the paper [43, pg. 1], ToonTalk provides “a series of interactive puzzles in a game-like narrative context”. Programming constructs and techniques are taught while children solve these puzzles. Referring to the examples in [43, pg. 3], message passing is accomplished by “giving a box or pad to a bird”. A mouse banging a hammer to merge two values represents simple operations such as addition in ToonTalk. Item copying is performed with “a magician’s wand”. Similarly there

[4] Visual programming languages use graphical elements or figures to code a program.

are other visual programming languages which help children to learn programming such as the Alice programming language [41] and Kodu from Microsoft [42].

There have been few educational or research games which used graphical elements to process human player inputs and solve research problems. One notable example could be FoldIt [29] where players predict protein structure using graphical biological components. The purpose of FoldIt is to crowd-source the work on protein folding which could not be effectively accomplished by software. Similarly, the purpose of our research game is to crowd-source code obfuscation strategies using a video game. Hence this game has a two-fold objective; the normal goal is to win in the game and the abstract goal is to generate obfuscated code.

2.4 Summary

In this chapter, we discussed about various forms of code obfuscation as well as other plausible methods to obfuscate code. Conventional code obfuscation requires programmed obfuscators which modifies an understandable code to an obfuscated version using different code obfuscation approaches [1]. Some examples of code obfuscation approaches that had been discussed are dead code insertion and using redundant operations to obfuscate mathematical expressions. We also observed that optimizing compilers [11, pg. 383] and pseudo-random number generators [3] could be used to obfuscate code.

We also reviewed the crowdsourcing concept and various models of crowdsourcing used in the field of business and science. We discussed about the idea of microwork and its examples such as Amazon's Mechanical Turk [25, 26] and iStockphoto [30]. The concept of crowd sourcing has been adapted to solve research problems as evident in FoldIt [29] and Duolingo [28]. There are research projects such as reCAPTCHA [27], which are used for web security purposes as well as a crowd sourced technique to digitize books. On the other hand, crowd sourcing has also been used by spammers to circumvent the security provided by CAPTCHA images [34].

Chapter 3

Developing code obfuscating games

There are several ways of obfuscating code using different transformation techniques as outlined in the “Related Work” chapter. The new method proposed in our work uses a human player playing a video game as the obfuscator (which converts source code to obfuscated form) as in Figure 3.1.

How could human players become obfuscating agents? Our focus was to analyze novel ways to represent obfuscating transformations by player interactions with game elements. While coming up with game strategies, the challenge that we faced was to decide the level of abstraction of information presented to the player. For example, should the operations be made obvious to the player, or should they be represented surreptitiously. After reflecting on a few game ideas, we decided to settle for the latter strategy as it would provide more flexibility for game design and also hide the coding process from the player. Our experiments with game ideas and subsequent analysis of whether they are suitable for this work is discussed in detail in the later sections.

The first proof of concept we made is a video game developed using GameMaker[1] software. Developing the game from the beginning gave us flexibility over game design. We created a game layout, where game elements represented the code elements such as variables or constant values. The interaction between the player character and the game elements represented the code operations such as arithmetic operations in the game. The constraints of the gameplay were such that the game generated the source code in the appropriate order such that the functionality of the code

[1]GameMaker 8.1 Standard Version 8.1.141 was used for developing this game.

was retained. Some game elements were introduced in the game to add obfuscating transforms to the source code during gameplay.

We were able to produce obfuscated code output by playing the game. Another challenge was to vary the obfuscated code output each time the game is played. Hence, we added a procedural content generation (PCG) technique [44] to randomly vary the game level. Subsequently the obfuscated code output varied with every instance of the game level.

This chapter is organized as follows: Section 3.1 presents conceptualization of the game idea; Section 3.2 describes implementation of the game, followed by a discussion about the evaluation results of obfuscation and a brief summary in Sections 3.3 and 3.4.

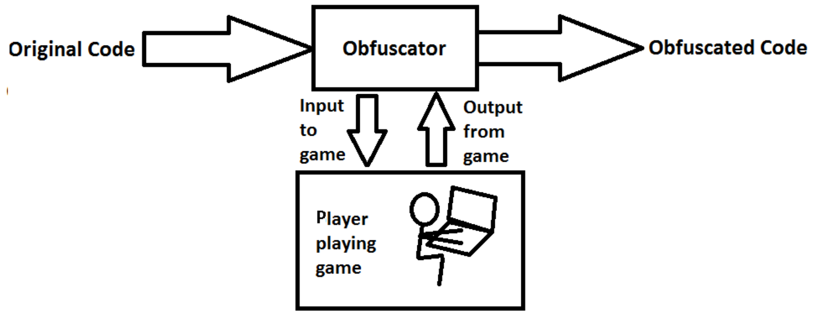


Figure 3.1: Code obfuscation with a video game

3.1 Conceptualization of game idea

As discussed by Alexander Zook and Mark O. Riedl [37], a game concept starts with a “vague specification” and then final design is built over “incremental development” through prototyping. The purpose of prototyping is to test the game mechanics and design ideas before the real implementation [38]. There are many techniques for prototyping game ideas. Some examples discussed in [38] include paper prototypes, idea sketches, and software prototypes. Conceptualizing a game idea is generally an iterative process with individual or group brainstorming sessions. There are several methods which are useful to conceive game ideas. A designer might list all ideas coming to mind on a certain topic, or use idea cards, or use random words and connect them with a story. For this game, we followed the design process using paper prototypes to analyze the playability of the game and feasibility of the idea [38]. This game was based on moving the player character across hanging platforms and popping colored balloons. For the prototype, we printed some colored balloons and drew the initial level in the game on a piece of paper. The game could be played by moving the colored balloons on the paper presentation of the level. The next step is to validate the idea based on the purpose of designing the game. It includes verifying whether the game is playable as well as technically feasible to develop.

The game ideas typically focus on the objective of the game which the players have to achieve while playing the game. The next step is to develop a story for the game which revolves around the goal or the objective. The procedure or game play is modeled according to the game objective. Game rules and resources are determined by the gameplay and the story of the game. Conflicts are added to the gameplay so that the player cannot achieve the goal directly.

3.1.1 The Idea

We had the flexibility to design the game from scratch. Hence we had the option to pick arbitrary game elements to represent code elements in the source code. Then we could design the game, adding constraints to the gameplay such that it generates the source code in sequential order. Taking a cue from ToonTalk [43], we started analyzing ideas with icons which could represent code elements as well as could be used in a game. After some trial and error, we finalized the idea of developing a game based on colors. Why colors? It was a design choice we made and there could be several other ways to achieve the same objective. Our thinking focused on ideas to relate colors to simple math operations. Later we planned to extend the concept to represent higher level loop statements. As shown by Farhad Mavaddat and Behrooz Parhami in [47], it is possible to design a one instruction set computer (OISC)[2] with math operations. This was the reason that we decided to start with simple math operations. Mavaddat and Parhami also prove that such a simple computer is Turing-complete since it could be used to execute complex operations such as conditional branching and loading and moving to memory addresses [47].

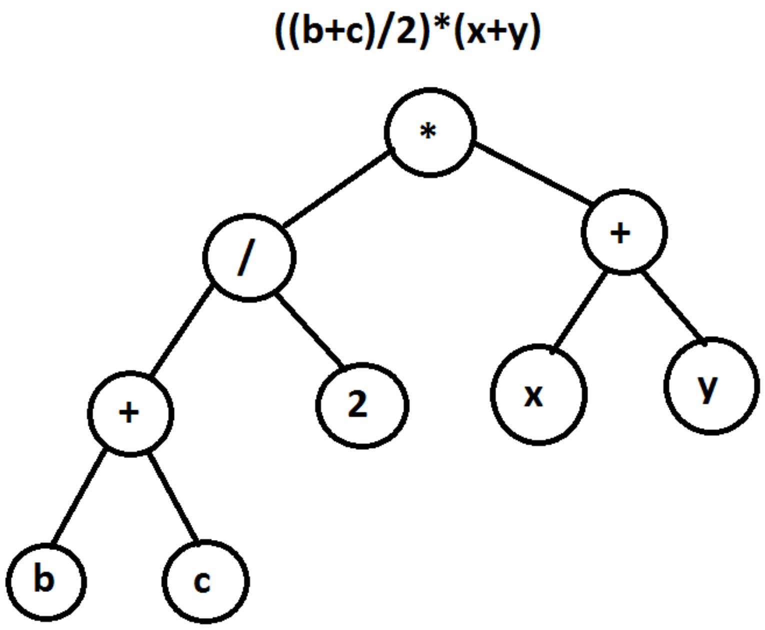


Figure 3.2: Abstract syntax tree for an arithmetic expression

We proceeded with creating an abstract syntax tree (AST) representation [45] of an arbitrary arithmetic expression as in Figure 3.2. The expression is segregated into binary operations and then a binary tree representing the AST form is drawn. The only reason to use binary operations was to prove the concept with basic operations and later extend it for other types of operations. The leaf nodes represent the operands whereas the inner nodes represent the operator in the binary operation. We decided to represent all nodes, leaf nodes as well as inner nodes, with different colors

[2]One instruction set computer (OISC) uses a single instruction for all types of computations.

(Figure 3.3).

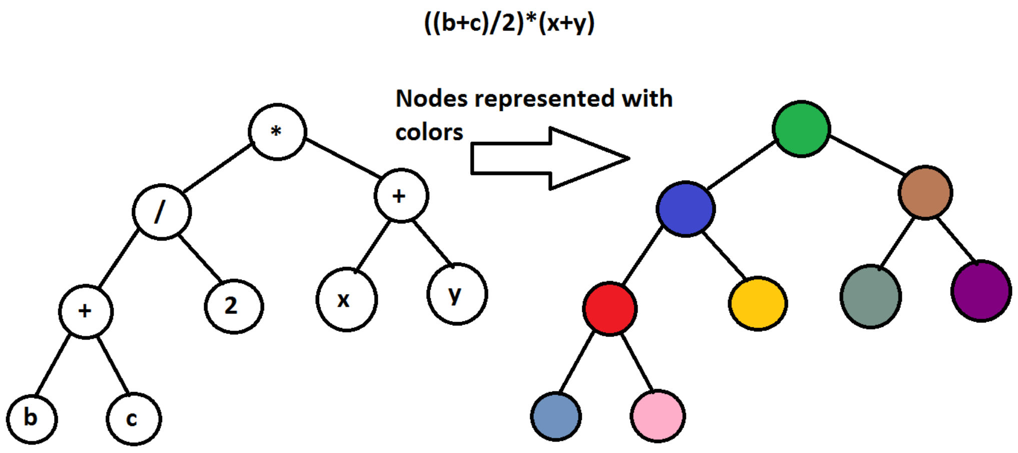


Figure 3.3: Abstract syntax tree to balloon mapping

The next step was how to represent a binary operation in the game. Since our purpose was to abstract the operation, we could use a two input machine, for example (Figure 3.4). The machine has two input channels: two colored balloons represent the operands in its input channels. It generates a colored output balloon representing the output of the binary operation. Initially these machines were one of the game elements in the prototype but later they were removed to make the game easier to understand.

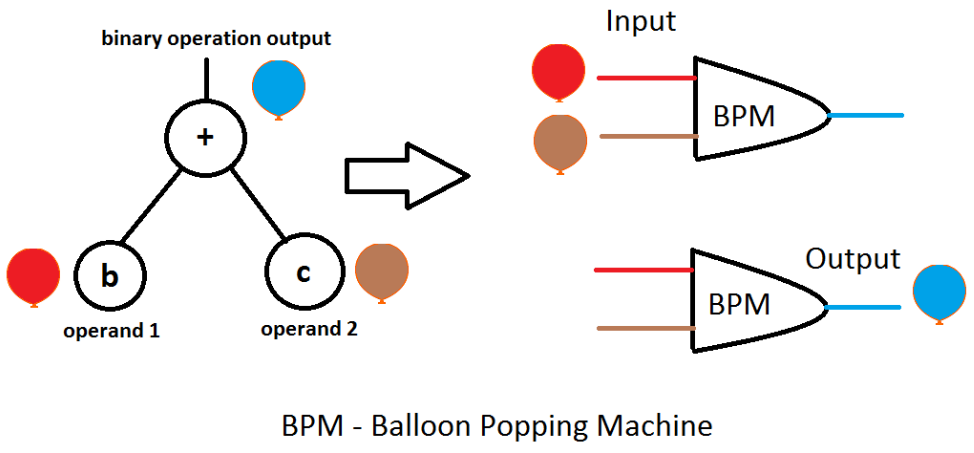


Figure 3.4: Binary operations with balloons

The subsequent step was to decide the obfuscating transformations to be used for the game. C.S. Collberg, C. Thomborson, and D. Low outline different code obfuscation approaches [1]. We selected some of them from the *computation transformation* category for this research game - inserting dead code, using "redundant operands", and code reordering. "Dead or irrelevant" code refers to junk code inserted into the source code that is not used for any required computations in the program. Transformations such as "redundant operands" added to math operations which are not required, such as a subexpression that computes a constant value and is not used in the actual calculation. As emphasized by Collberg [1], there are different forms of obfuscation to abstract "real control-flow" in the program. Code reordering involves changing the order of independent sections of the code as shown in the code samples shown in Listings 3.1 and 3.2. Spaghetti code involves splitting the program code into sections and using GOTO statements to transfer flow to different sections, making the program flow difficult to understand. We will use "junk code" to mean "dead or irrelevant" code insertion obfuscation, "redundant operands" to mean redundant operand obfuscation, and "code re-ordering" or "spaghetti code" to mean control flow obfuscation for the rest of the thesis. The process by which these obfuscating transforms were derived from the game is discussed in the technical implementation section of this chapter.

```
void calcExpr(int b,int c
   ,int x,int y)
{
 int r1=b+c;
 int r2=r1/2;
 int r3=x+y;
 int res=r2*r3;
}
```

Listing 3.1: Original code

```
void calcExpr(int b,int c
   ,int x,int y)
{
 int r3=x+y;
 int r1=b+c;
 int r2=r1/2;
 int res=r2*r3;
}
```

Listing 3.2: Re-ordered code

Finally, the game idea revolved around using different colors to represent the code elements. The next step was to introduce game elements to represent the colors and obstacles, and to develop a premise or story for the game. A simple representation of the idea is outlined in Figure 3.5. First different colors are selected for corresponding nodes in the AST. Then the operands are assigned colored gems and the operations are shown with multicolored tiles with a colored balloon.

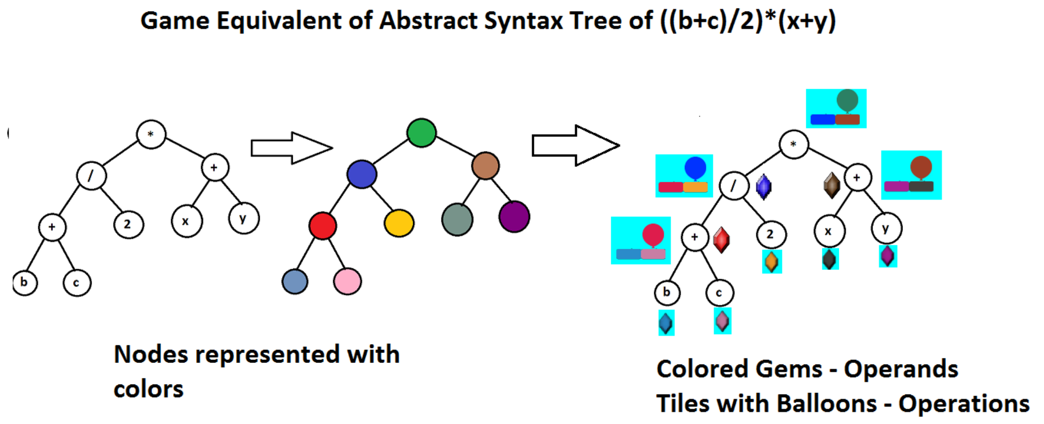

Game Equivalent of Abstract Syntax Tree of ((b+c)/2)*(x+y)
*
/
+
2
x
y
b
c
Nodes represented with colors
Colored Gems - Operands
Tiles with Balloons - Operations

Figure 3.5: Evolution of AST to game elements

3.1.2 The Game Structure

Premise. This game is a platform game [46] where the player controls a game avatar (the player character) jumping over suspended platforms and surviving obstacles to solve the puzzle in the game. In this game, the avatar is a small boy. The game starts when the player character enters the maze of suspended platforms, colored gems and colored balloons on multicolored tiles as shown in Figure 3.6. The goal of the player is to explore the maze and pop the colored balloons on the multicolored tiles, but at the same time to survive the roaming ghost and jumping monkeys. Hence the game is named “Popper”.

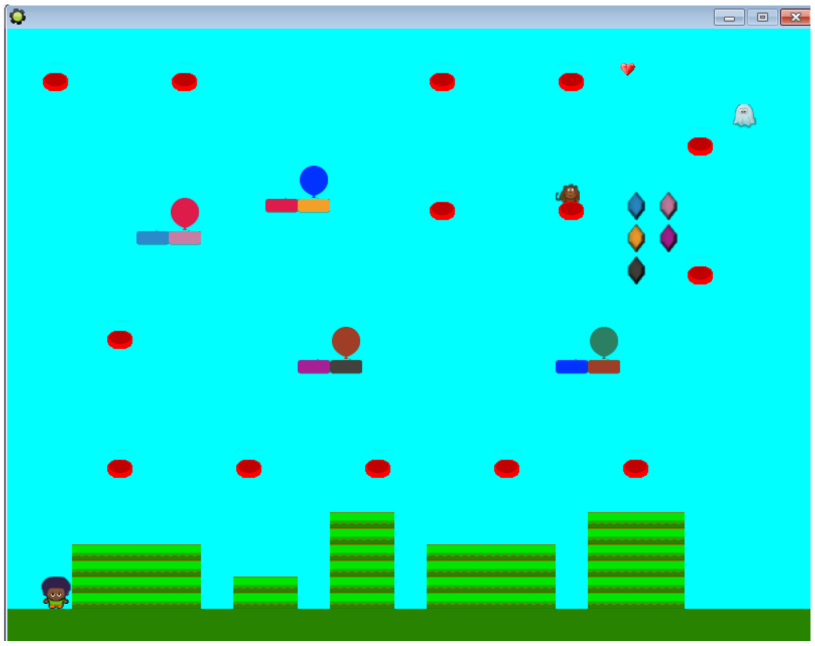


Figure 3.6: Sample game level

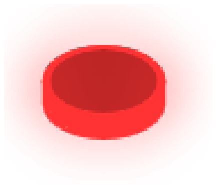
Game elements. The player character has the avatar of a small boy as shown in Figure 3.7d[3]. He can move left or right, jump, and climb ladders. He loses a life for every collision with the roaming ghost (Figure 3.7c). He is moved back to the starting position in the level for each collision with the jumping monkey (Figure 3.7a). When

[3]All art assets for this game have been taken from OpenGameArt.Org (http://opengameart.org/). See Appendix C for license information.

he bumps into one of the neon tokens (Figure 3.7b), he may win a life as in Figure 3.8.

Non-player characters (NPCs) are added to introduce conflicts in the game. The roaming ghost is an NPC (Figure 3.7c). It moves randomly across the screen. It is harmless except when it collides with the player avatar, reducing the number of lives available to the player. Other NPC characters are the jumping monkeys (Figure 3.7a) who appear on multicolored tiles and jump up and down with varying speeds (Figure 3.10). They make it difficult for the player avatar to get near the colored balloons. If the player avatar is hit by a jumping monkey, he is moved back to the initial position at the start of the game.

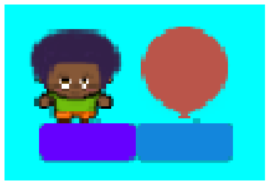
(a) Jumping monkey

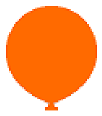

(b) Neon tokens

(c) Roaming ghost

(d) Player character

(e) Ladders


(f) Multicolored tile


(g) Colored balloons

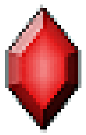

(h) Gems

Figure 3.7: Game elements

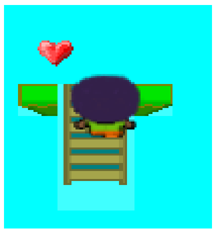


Figure 3.8: Player earning a life

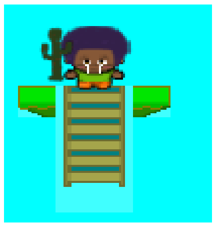


Figure 3.9: Player expression on hitting the cactus plant

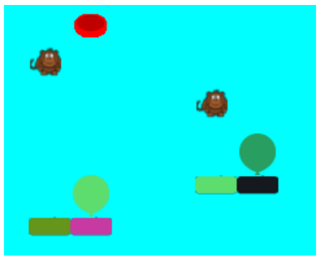


Figure 3.10: Monkey jumping on multicolored tiles

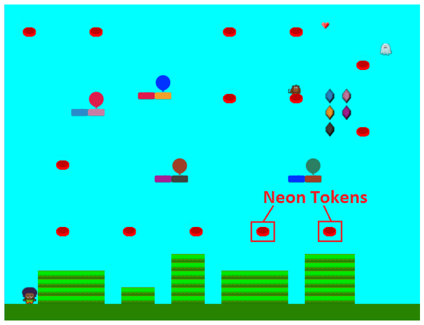


Figure 3.11: Neon tokens in a sample game level

The neon tokens as in Figure 3.7b are scattered throughout the suspended platforms in the game (Figure 3.11). They act as possible power-ups for the player character, and have twofold use. When the player avatar collides with them, they transform into a ladder (there are two types of ladders - refer to Figure 3.7e) for him to climb and get access to nearby multicolored tiles. Also a chance-based system internally determines if this ladder will contain a cactus plant or a heart which is a life scored (Figures 3.8 and 3.9).

A few colored gems will be arranged together in the game screen (Figure 3.7h). They provide the clues about which multicolored tile should be the next destination for the player avatar. The strategy depends on matching the colors of the gem with the colors of the tiles.

The multicolored tiles (Figure 3.7f) with colored balloons (Figure 3.7g) in the game form the core of the platformer game. A multicolored tile is painted with two different colors in the game. If the player can identify the correct tile, then he or she can move the player avatar to reach that tile by jumping through surrounding tiles and ladders. Once on the tile, the player avatar collides with the balloon on that tile and pops the balloon.

Game Mechanics. This game belongs to the genre of action games. This is a platformer game with multicolored tiles acting as mid-air platforms, green blocks, colored balloons, gems, neon tokens and obstacles like ghosts and monkeys, forming the puzzle (Figure 3.6) which is produced by a procedural content generator (PCG) [44]. The basic mechanic in this game, as in other platform games, is jumping between hanging platforms. The goal of the game is to pop all the balloons on the multicolored tiles or platforms. As already discussed briefly, the player avatar jumps between different multicolored tiles to reach the correct tile based on the colored gems present in the game screen.

The color matching rule between the gems and multicolored tiles is illustrated in Figure 3.12. Based on the colors of gems available, a multicolored tile is picked by the player. That means the multicolored tile should have the same colors as any two of the gems available. Noticeably, in Figure 3.12, the player could either select the multicolored tile shown in Step 1 or the tile shown in Step 2. Both tiles satisfy the rule of color matching using the available gems at the start of the game. This is later used for control flow obfuscation such as code re-ordering.

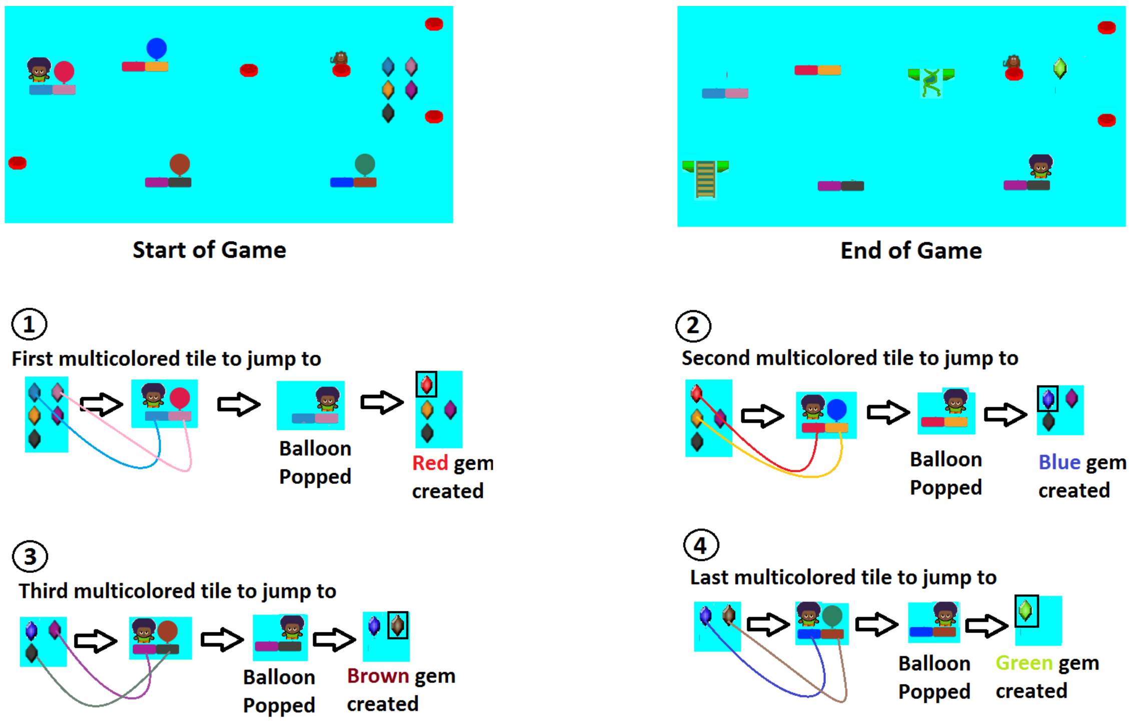


Figure 3.12: Color matching rule in the game

While trying to jump to the required multicolored tile, the player avatar may come across a few hurdles. If the avatar collides with the roaming ghost it will reduce the number of lives available to the player. A good survival strategy for the player is to move quickly away from the path of the roaming ghost. In case a life is lost, it can be recovered by using one of the neon tokens. Jumping on the neon tokens will provide a ladder to help with vertical movement in the game as well as to win a life or a cactus plant.

Jumping monkeys may be present on the tile the player is moving to. In that case, the player has to be careful so that the avatar does not collide with the monkey. In that case, the player will have to trace the entire route to the tile again from the starting point in the game. The jumping monkeys have varying speeds. They jump slowly and then increasingly accelerate their speed. After reaching the maximum acceleration or jump speed, the cycle is repeated again. One strategy to beat the jumping monkeys is to wait for the slow speed motion and move quickly to pop the balloon.

Since this was a research game being developed as a proof-of-concept, a few game elements have been skipped. For example, there is no reward system integrated in the game. The neon tokens are introduced in the game to make the goal achievable for the player. These tokens help the avatar gain vertical advantage in the game as shown in Figure 3.13.

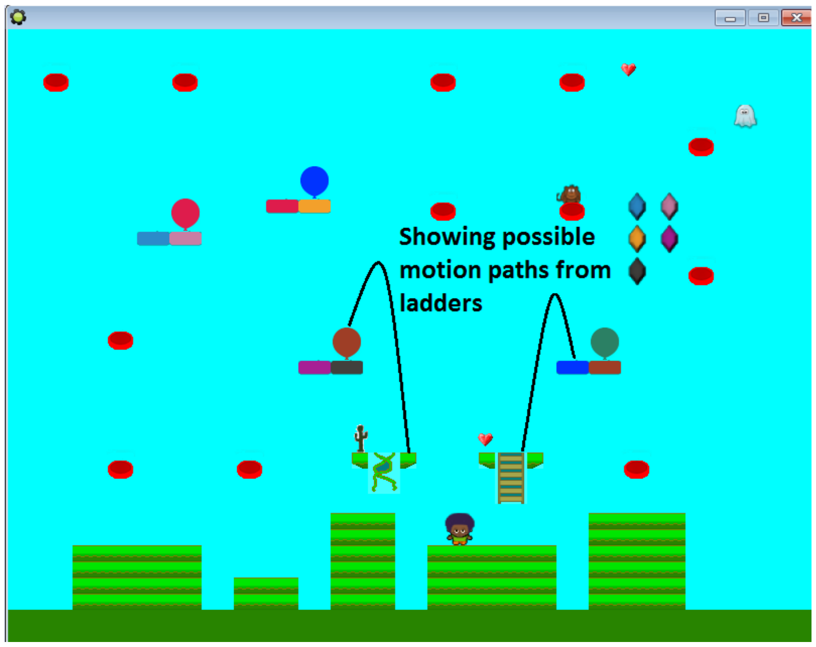


Figure 3.13: Motion advantage by neon tokens

There are elements such as hearts on ladders which change the game state. That is, acquiring a heart increases the number of lives available to the player. A cactus plant does not contribute to the game state except for showing an explicit emotion by the player avatar.

At the end of the game, there will be no balloons left on the multicolored tiles as shown in Figure 3.14. If the player loses all lives before popping all balloons, the game starts over.

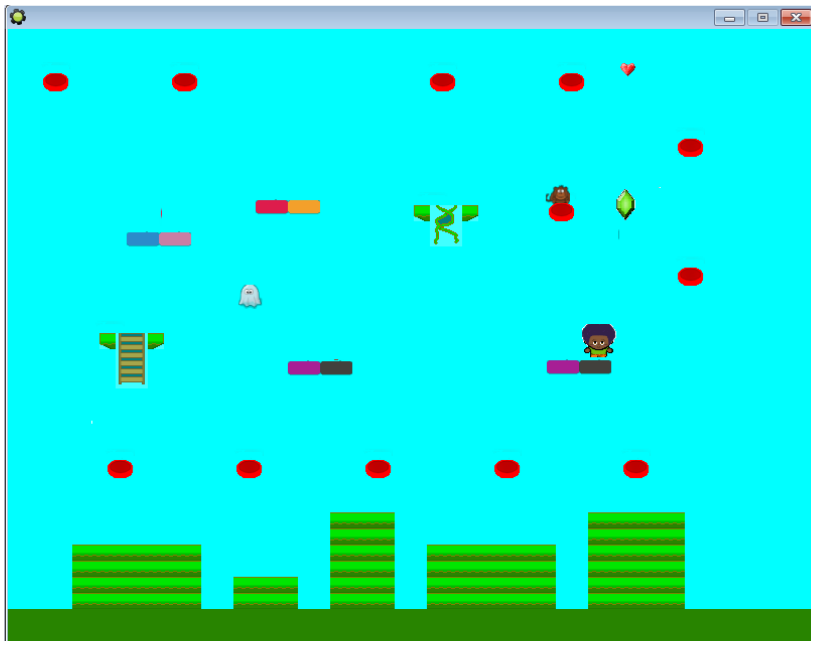


Figure 3.14: The end of the game

3.2 Technical illustration of the game

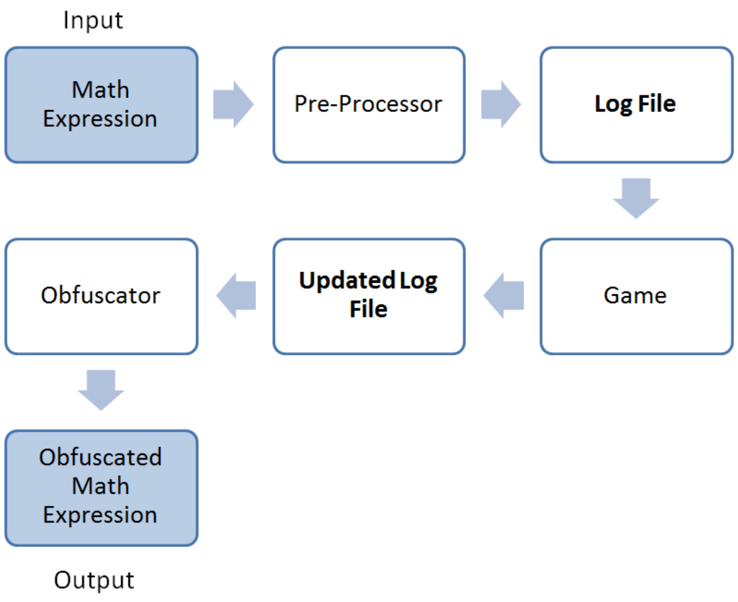


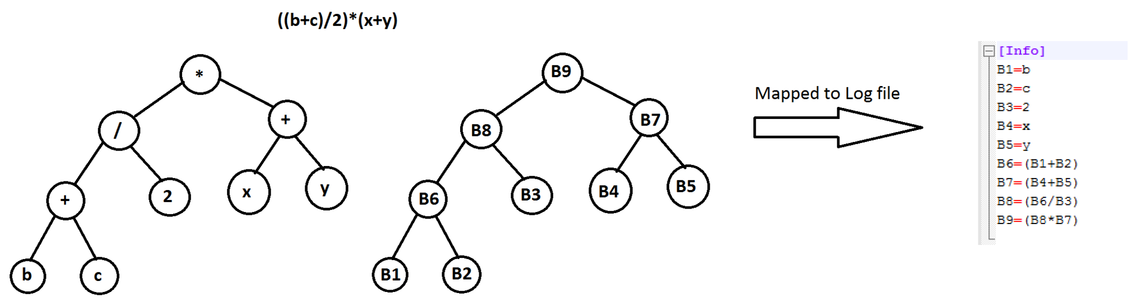
Figure 3.15: The process flow

In this section, we will discuss the technical implementation of obfuscating math expressions with the developed game. A simple program flow of the process is shown in Figure 3.15. The implementation process is explained with the help of the math expression $((b + c)/2) * (x + y)$. As shown in Figure 3.15, the math expression is processed by a "pre-processor" script which produces a log file. The game we developed in GameMaker reads the log file and generates the appropriate game level with game elements representing the code elements. Once the game level is played, the game generates an updated log file which is then read by an "obfuscator" script. This script generates the obfuscated output based on the player inputs collected from the game. A detailed explanation of each segment of the implementation is discussed in the following sections.

3.2.1 Pre-processor

The pre-processor is a Python script which splits up the math expression and arranges the segments in a log file to be processed by the game. The game will read the information in the log file and design the game level to represent the math expression using game elements.

The math expression $((b + c)/2) * (x + y)$ is converted to an abstract syntax tree (AST) [45]. The nodes (leaf nodes and operator nodes) are assigned numbered identifier names with the prefix "B" such as B1, B2, B3, etc. Conceptually, an operator node gets converted to a leaf node when the binary operation corresponding to it is performed. The order in which the nodes are labeled is first leaf nodes, then first-order nodes, then second-order nodes until the topmost node in the binary tree is reached (Figure 3.16). The corresponding mapping of the identifiers with variables, constants and operation in the math expression is logged in the log file as shown in Figure 3.16.



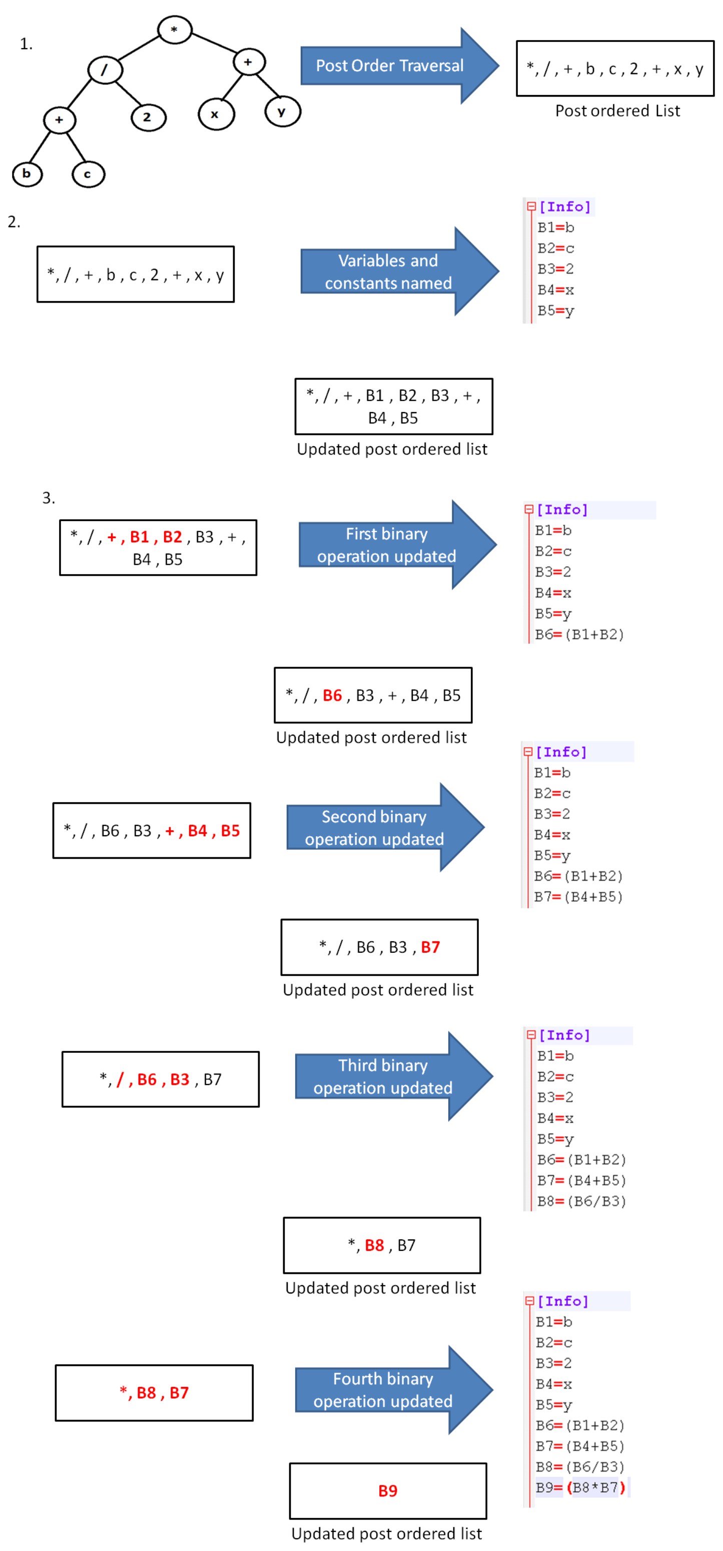
```
[Info]
B1=b
B2=c
B3=2
B4=x
B5=y
B6=(B1+B2)
B7=(B4+B5)
B8=(B6/B3)
B9=(B8*B7)
```

Figure 3.16: Labeling of nodes in abstract syntax tree

In the pre-processor script, the math expression is first parsed and ordered in a list using a post-order traversal of the AST. Then the leaf nodes (variables) are extracted and updated in the log file. In the next step, the binary operations from the post-ordered list of code elements are identified and updated in the log file. First each element in the list is checked if it is an operator. If yes, then the next two elements in the list is checked if they are not operators. If yes, then the binary operation is updated. The pseudo-code for the pre-processor script is given in Algorithm 1 and its steps with the example AST are shown in Figure 3.17 (page 38).

Algorithm 1 Pre-processor

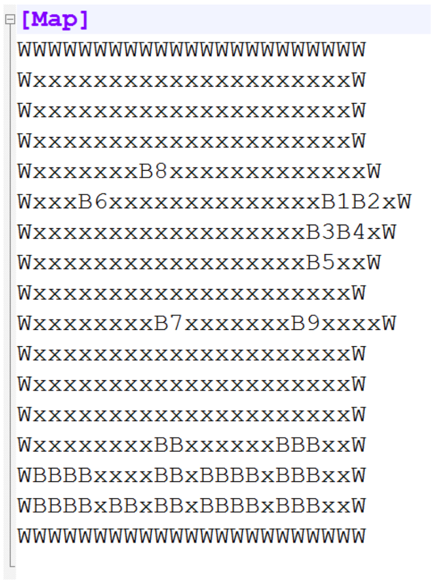
```
1: procedure PREPROCESSOR($math\_expr$)
2:     Parse $math\_expr$ in postorder and generate a list of code elements as $postorderlist$
3:     Create an identifier count $iden\_count$
4:     for each code element $c$ in $postorderlist$ do
5:         if $c$ is digit or alphanumeric then
6:             Create an identifier name with “B” and $iden\_count$
7:             Assign $c$ to the newly created identifier name
8:             Increment $iden\_count$
9:             Update log file “Info” section with identifier name and $c$
10:            Replace code element $c$ in $postorderlist$ with identifier name
11:        end if
12:    end for
13:    while there is no operator in $postorderlist$ do
14:        for each code element $c$ in $postorderlist$ do
15:            if $c$ is an operator then
16:                if next two elements in $postorderlist$ are not operators then
17:                    Create an identifier name with “B” and $iden\_count$
18:                    Update the binary operation with $c$ as operator and next two
                       elements in $postorderlist$
19:                    Assign the binary operation to the newly created identifier
                       name
20:                    Increment $iden\_count$
21:                    Update the binary operation with $c$ as operator and next two
                       elements in $postorderlist$
22:                    Assign the binary operation to the newly created identifier
                       name
23:                    Increment $iden\_count$
24:                    Update log file “Info” section with the identifier name and the
                       binary
                       operation
25:                    Replace the operands and operator of the binary operation in
                       $postorderlist$ with the identifier name
26:                end if
27:            end if
28:        end for
29:    end while
30: end procedure
```



1.
Post Order Traversal
*, / , + , b , c , 2 , + , x , y
Post ordered List
2.
*, / , + , b , c , 2 , + , x , y
Variables and constants named
[Info]
B1=b
B2=c
B3=2
B4=x
B5=y
*, / , + , B1 , B2 , B3 , + , B4 , B5
Updated post ordered list
3.
*, / , + , B1 , B2 , B3 , + , B4 , B5
First binary operation updated
B6=(B1+B2)
*, / , B6 , B3 , + , B4 , B5
Updated post ordered list
Second binary operation updated
B7=(B4+B5)
*, / , B6 , B3 , B7
Updated post ordered list
Third binary operation updated
B8=(B6/B3)
*, B8 , B7
Updated post ordered list
Fourth binary operation updated
B9=(B8*B7)
B9
Updated post ordered list

Figure 3.17: Steps of preprocessor with the example AST

The pre-processor also includes a simple procedural content generation engine [44]. The pseudocode for this procedural content generation script is provided in Appendix 6. This engine determines the position of different game elements such as the green blocks, the gems, and the multicolored tiles with colored balloons. The engine relies on basic distance calculation of neighboring elements while deciding the position of the elements. Figures 3.18 (page 39), 3.19 and 3.20 show the output of the engine which is updated in the log file read by the game. The exact coordinates of the game elements are calculated and logged in the log file to reduce processing effort in the game.



```
[Map]
WWWWWWWWWWWWWWWWWWWWWWW
WxxxxxxxxxxxxxxxxxxxxxW
WxxxxxxxxxxxxxxxxxxxxxW
WxxxxxxxxxxxxxxxxxxxxxW
WxxxxxxxB8xxxxxxxxxxxxxW
WxxxB6xxxxxxxxxxxxxxB1B2xW
WxxxxxxxxxxxxxxxxxxB3B4xW
WxxxxxxxxxxxxxxxxxxB5xxW
WxxxxxxxxxxxxxxxxxxxxxW
WxxxxxxxxB7xxxxxxxB9xxxxW
WxxxxxxxxxxxxxxxxxxxxxW
WxxxxxxxxxxxxxxxxxxxxxW
WxxxxxxxxxxxxxxxxxxxxxW
WxxxxxxxxBBxxxxxxBBBxxW
WBBBBxxxxBBxBBBBxBBBxxW
WBBBBxBBxBBxBBBBxBBBxxW
WWWWWWWWWWWWWWWWWWWWWWW
```

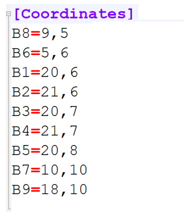
Figure 3.18: Procedural content generation map

W = Wall

B = Green Blocks

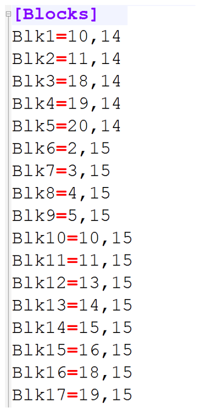
B1,B2,B3,B4,B5 = Gems

B6, B7, B8, B9 = Multicolored tiles with balloons



```
[Coordinates]
B8=9,5
B6=5,6
B1=20,6
B2=21,6
B3=20,7
B4=21,7
B5=20,8
B7=10,10
B9=18,10
```

Figure 3.19: Coordinates of game elements in the PCG map



```
[Blocks]
Blk1=10,14
Blk2=11,14
Blk3=18,14
Blk4=19,14
Blk5=20,14
Blk6=2,15
Blk7=3,15
Blk8=4,15
Blk9=5,15
Blk10=10,15
Blk11=11,15
Blk12=13,15
Blk13=14,15
Blk14=15,15
Blk15=16,15
Blk16=18,15
Blk17=19,15
```

Figure 3.20: Coordinates of green blocks in the PCG map

3.2.2 Processing of log file

The game interprets the log file generated by the pre-processor starting with the "Info" section (Figure 3.16). This section provides details about the number of gems and number of multicolored tiles to be created for the game level. The gems and multicolored tiles are placed according to the coordinates from the "Coordinates" section of the log file as created by the PCG engine. The neon tokens are placed randomly across the game screen.

The gems in the game represent the identifiers which have variables or constants as their value such as B1, B2, B3, B4 and B5 in the running example (Figure 3.16). Random colors are picked for each gem using the random function of the GameMaker scripting language (see Figure 3.21 on page 41). The arrangement of the gems in the game is shown in Figure 3.22. The "Coordinates" section of the log file is used to decide the position of the gems on the game screen.

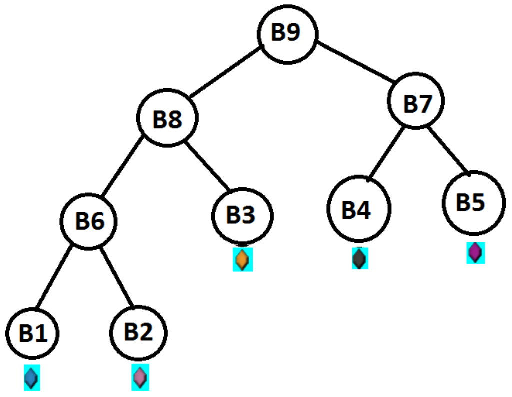


Figure 3.21: Random gems assigned to variables and numbers

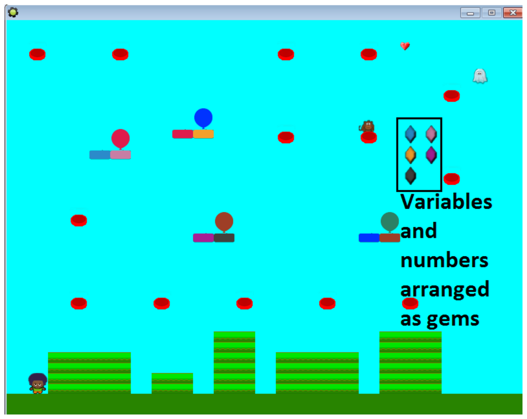


Figure 3.22: Gems

The multicolored tiles with colored balloons represent the identifiers which denote the binary operations in the math expression such as B6, B7, B8 and B9 in Figure 3.16. The operands in the binary operation (the colored gems) decide the colors of the tile. The color of each balloon is randomly selected using the random function of the GameMaker scripting language (Figure 3.23). A graphical representation of the tiles is shown in Figure 3.24.

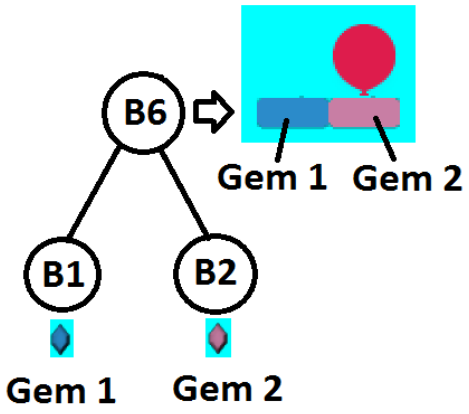


Figure 3.23: Multicolored tile assigned to binary operations

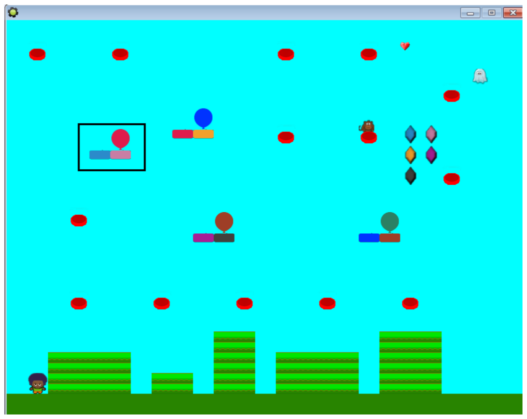
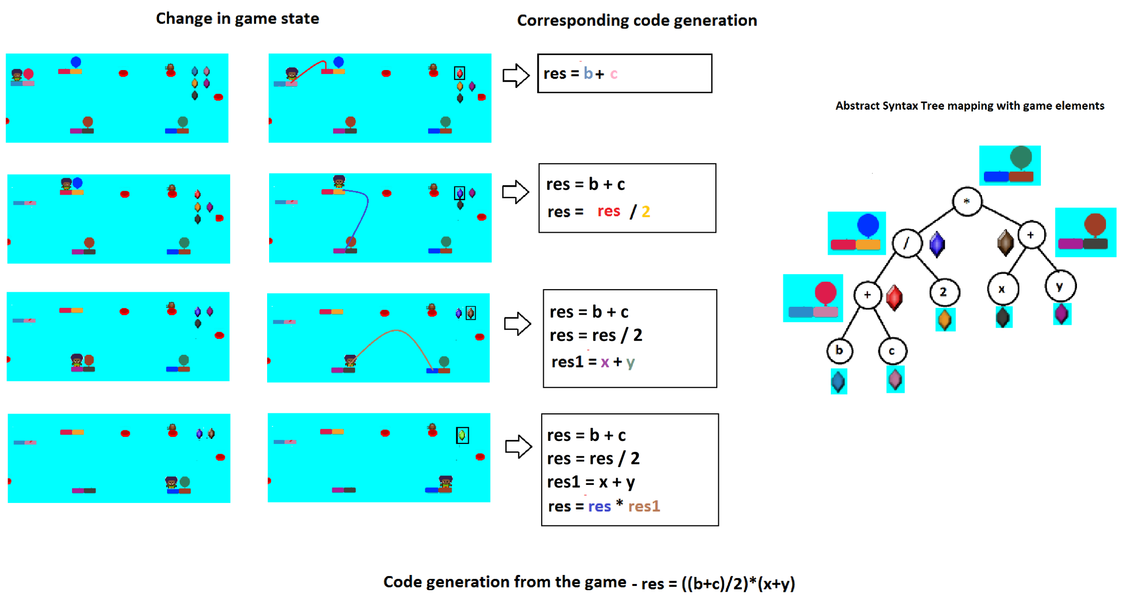


Figure 3.24: Tiles

Multiple instances of neon tokens are spawned randomly across the game screen.

The position of the neon tokens are chosen so that they do not overlap any other static game elements such as colored gems or multicolored tiles. The neon tokens are one of the game elements used for obfuscating transforms. The collision of dynamic game elements such as ghosts with neon tokens will not result in any change of the game state.

The code generation resulting from the game being played is shown in Figure 3.25.


Change in game state
Corresponding code generation
Abstract Syntax Tree mapping with game elements
*
/
+
+
2
x
y
b
c
Code generation from the game - res = ((b+c)/2)*(x+y)

```
res = b + c
```

```
res = b + c
res = res / 2
```

```
res = b + c
res = res / 2
res1 = x + y
```

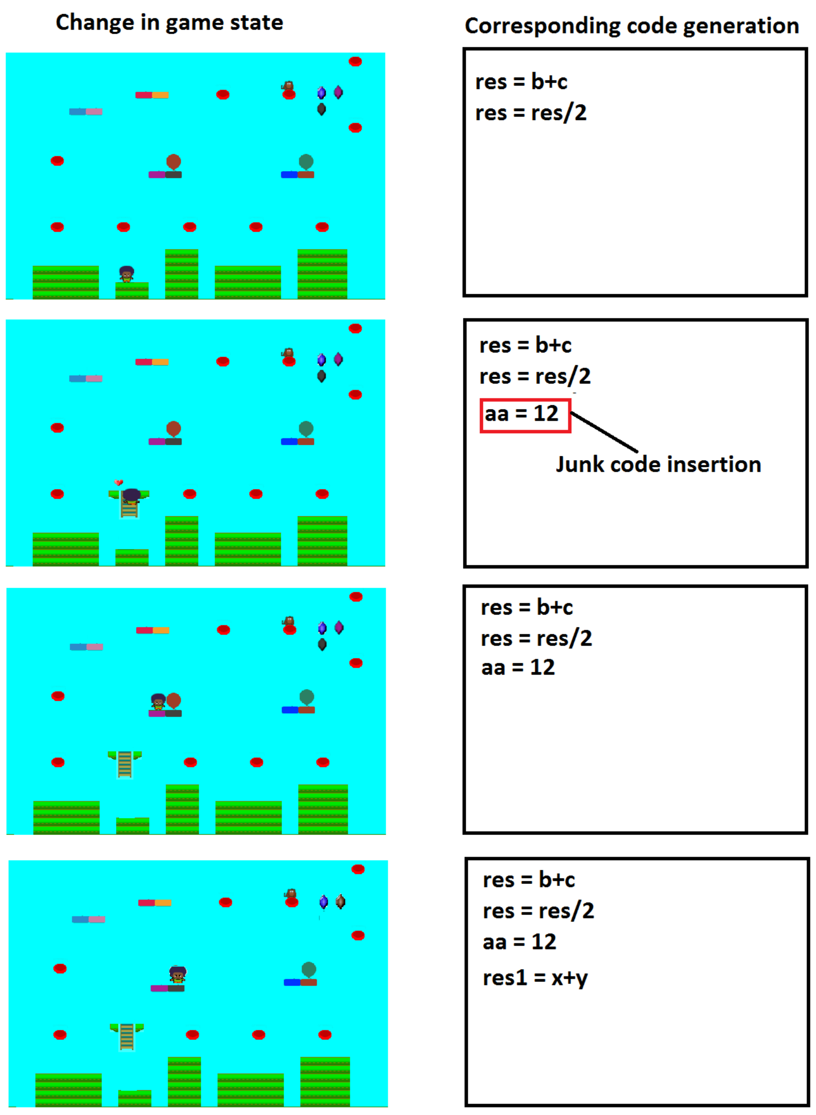
```
res = b + c
res = res / 2
res1 = x + y
res = res * res1
```

Figure 3.25: Code generation in the game

3.2.3 Obfuscating transforms

We briefly described the obfuscating transforms that this game could simulate in the “Idea” section of this chapter. They were junk code insertion, adding redundant operands to binary operations, and code reordering. The obfuscation process in this game is based on the player avatar movement in the game induced by the various situations in the game. Therefore player avatar positions at different times in the game are recorded in the log file. The data are later processed by the obfuscator script to produce the obfuscated output. The process by which these obfuscating transforms are derived from game elements and player interactions is explained in detail in this section.

The ladders spawned by neon tokens are used for junk code insertion. When the player avatar climbs up the ladders, the relative distance of the avatar between the previous position and the current position in x and y coordinates is calculated and added. This value is assigned to an arbitrary variable and inserted into the source code as shown in Figure 3.26 (on page 44). In this figure, the player avatar jumps and collides with the neon token above his head. This provides him a ladder to climb and reach the nearest multicolored tile. During the jump action of the player avatar, junk code gets added to the obfuscated code in the background. The result of calculating and adding the relative distance of the player avatar in x and y coordinate in this example is 12. This value is assigned to an arbitrary variable, “aa”, and added to the code being generated in the background. The arbitrary variables used in this game are randomly selected from a name pool (a list of variable names which does not include the variables in the math expression).



```
res = b+c
res = res/2
```

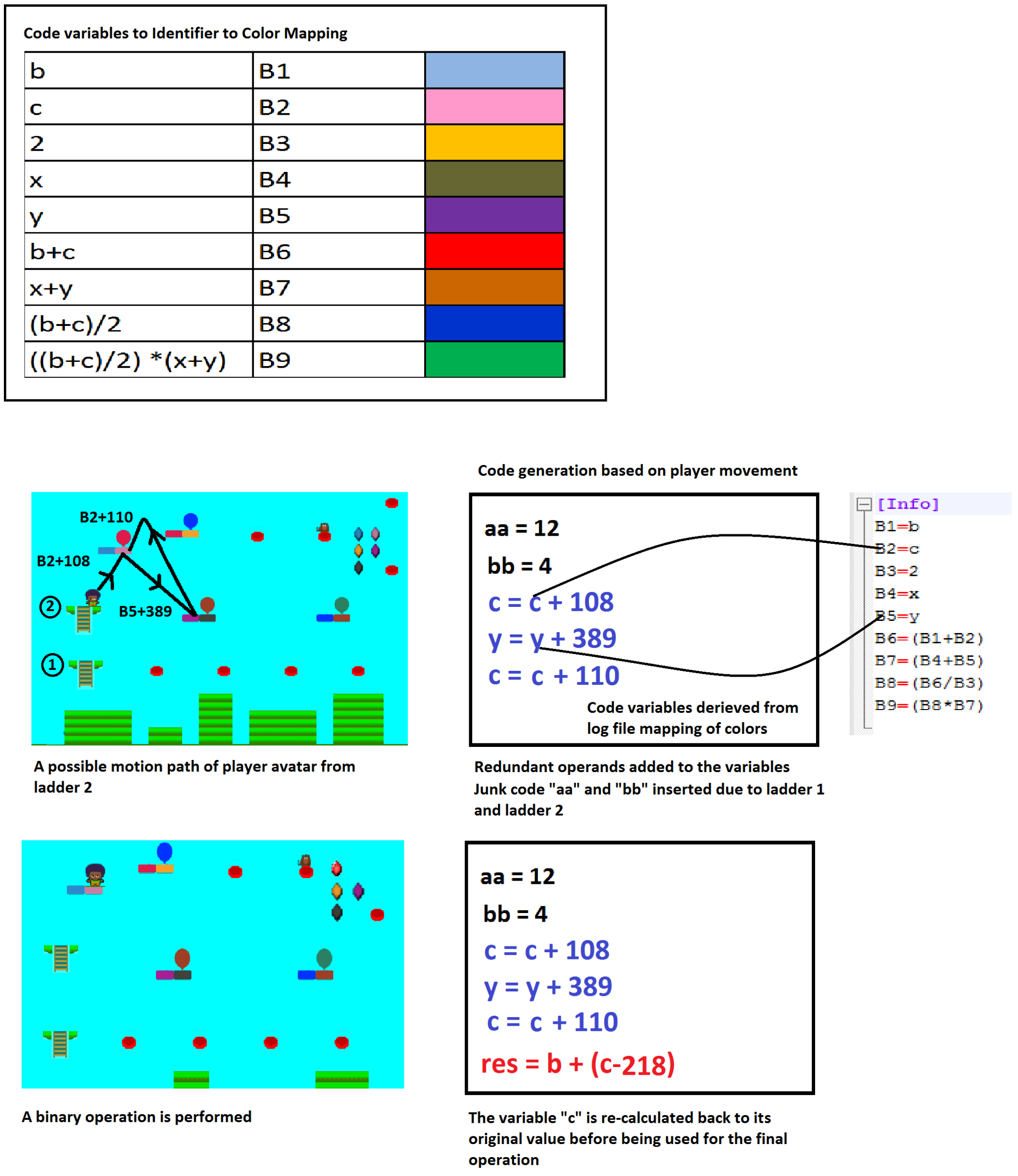
```
res = b+c
res = res/2
aa = 12
```

```
res = b+c
res = res/2
aa = 12
```

```
res = b+c
res = res/2
aa = 12
res1 = x+y
```

Figure 3.26: Junk code generation

During the course of the game, the player avatar will jump between different multicolored tiles to achieve the goals in the game. The colors in the tile represent the code elements in the math expression being obfuscated. When the player avatar lands on one of the tiles or hits it from below, the x-coordinate value of the avatar is recorded. That value is used as a redundant operand for the code elements represented by the tile. A multicolored tile consists of two colors, say color1 and color2. The essence of this game is to represent variables, constants, or operations with color. Hence a multicolored tile represents two code elements: say color1 represents element1 and color2 represents element2. The redundant operand is used with the code element represented by the colored part of the tile that the player avatar collides with. So if the player avatar collides with color1 of the tile, then the x-coordinate value of the avatar is used as a redundant operand value with element1. The current x and y coordinates of the player avatar are used to randomly choose the binary operation between the redundant operand value and the code element from a list of operations $(+, -, *, /)$. While creating the obfuscated version of the math expression, the value of the code element is re-calculated before it is used in the actual computation of the math expression, as shown in Figure 3.27.



Code variables to Identifier to Color Mapping

b	B1	🟦
c	B2	🟪
2	B3	🟨
x	B4	🟫
y	B5	🟪
b+c	B6	🟥
x+y	B7	🟧
(b+c)/2	B8	🟦
((b+c)/2) *(x+y)	B9	🟩

A possible motion path of player avatar from ladder 2

Code generation based on player movement

```
aa = 12
bb = 4
c = c + 108
y = y + 389
c = c + 110
```

Code variables derieved from log file mapping of colors

```
[Info]
B1=b
B2=c
B3=2
B4=x
B5=y
B6=(B1+B2)
B7=(B4+B5)
B8=(B6/B3)
B9=(B8*B7)
```

Redundant operands added to the variables Junk code "aa" and "bb" inserted due to ladder 1 and ladder 2

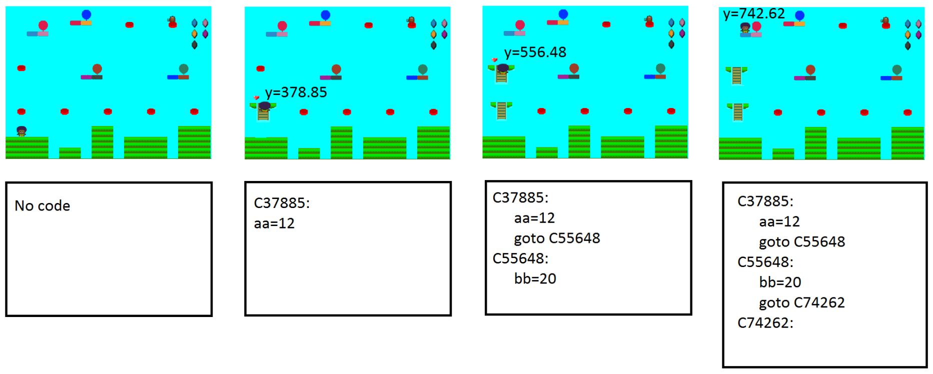
A binary operation is performed

```
aa = 12
bb = 4
c = c + 108
y = y + 389
c = c + 110
res = b + (c-218)
```

The variable "c" is re-calculated back to its original value before being used for the final operation

Figure 3.27: Adding a redundant operand for obfuscation

The y-coordinate value of the player avatar is also recorded in the game when a collision happens between the avatar and neon tokens or multicolored tiles. This value is later used for generating spaghetti code. A very simple illustration of control flow obfuscation such as spaghetti code generation is shown in Figure 3.28.



```
No code
```

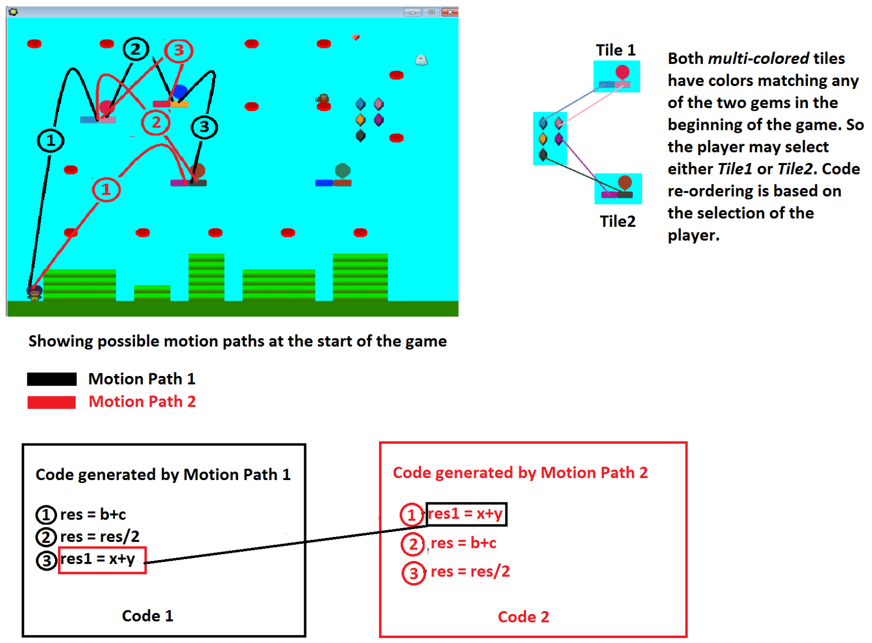
```
C37885:
aa=12
```

```
C37885:
    aa=12
    goto C55648
C55648:
    bb=20
```

```
C37885:
    aa=12
    goto C55648
C55648:
    bb=20
    goto C74262
C74262:
```

Figure 3.28: Spaghetti code generation for obfuscation in the game

Another control flow obfuscation, code re-ordering, is obtained by introducing different orders in which balloons could be popped. In Figure 3.29, there are two motion paths the player could follow to pop the first balloon in the game. Depending on the path selected by the player, code is re-ordered as shown in Figure 3.29. This method is useful when the math expression is pre-obfuscated. That is, instead of using a math expression in the form $((b + c)/2) * (x + y)$, we can manually (or via a script or automated method) obfuscate it into the form $((b+c)/2)*(x+y)*(1+0)$.


Tile 1
Tile2
Both *multi-colored* tiles have colors matching any of the two gems in the beginning of the game. So the player may select either *Tile1* or *Tile2*. Code re-ordering is based on the selection of the player.
Showing possible motion paths at the start of the game
▬ Motion Path 1
🟥 Motion Path 2
Code generated by Motion Path 1
Code 1
Code generated by Motion Path 2
Code 2

```
1 res = b+c
2 res = res/2
3 res1 = x+y
```

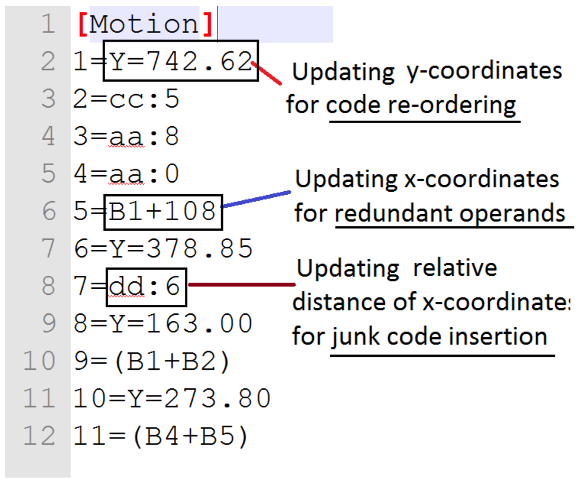
```
1 res1 = x+y
2 res = b+c
3 res = res/2
```

Figure 3.29: Code re-ordering for obfuscation

The game updates the log file generated by the pre-processor, with player inputs collected while playing the game level. These values are processed by the obfuscator script to generate the obfuscated math expression.

3.2.4 Output log file

The output log file contains the data from the game, needed for obfuscating transforms. A graphical illustration is shown in Figure 3.30. The "=" symbol is used as a delimiter in the log file.



```
[Motion]
1=Y=742.62
2=cc:5
3=aa:8
4=aa:0
5=B1+108
6=Y=378.85
7=dd:6
8=Y=163.00
9=(B1+B2)
10=Y=273.80
11=(B4+B5)
```

Updating y-coordinates for code re-ordering
Updating x-coordinates for redundant operands
Updating relative distance of x-coordinates for junk code insertion

Figure 3.30: Values used for code obfuscation updated in log file

3.2.5 Obfuscator

Our obfuscator script consists of Java classes which parse the log file and produce the final obfuscated output. The obfuscated output is generated as a C program. For simplicity, the obfuscator script uses a boilerplate C template which it updates with obfuscated code segments.

At first a new file is created and named "obfuscatedcode.c". The content of the C file template selected is updated in this new file. The variables used for junk code generation are defined as shown in Listing 3.3.

```
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 int aa,bb,cc,dd;
```

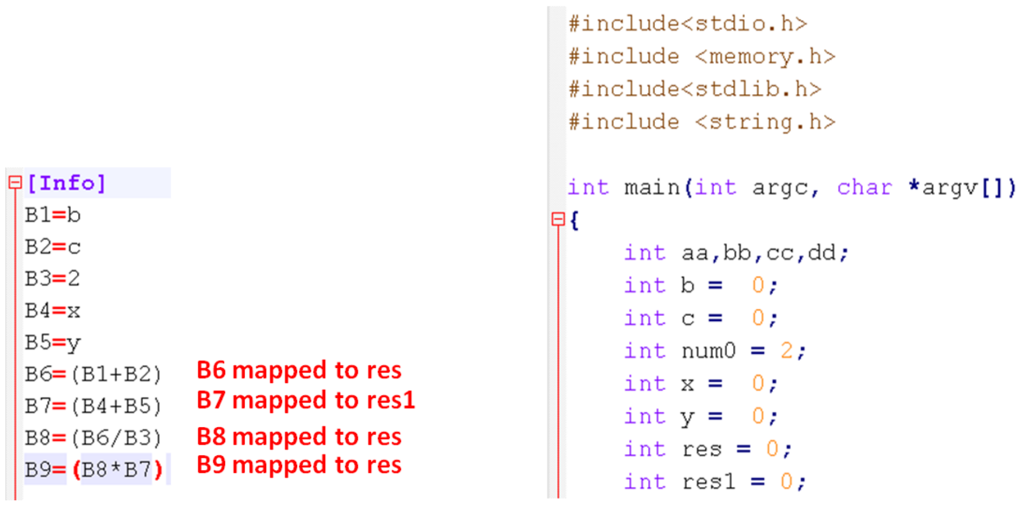
Listing 3.3: C code file with junk code variables

In the next step, the "Info" section of the log file is parsed and the variable definitions are generated. The math expression is represented in code as shown in Listing

3.4. The mapping of the variables is derived from the original code as shown in Figure 3.31.

```
#include <stdio.h>
int main()
{
 int b, c, x, y, res, res1;
 res = b + c;
 res = res /2;                        //res = (b+c)/2
 res1 = x + y;
 res = res * res1;                    //res = ((b+c)/2) * (
    x+y)
 printf("Calculated value = %d\n",res);
 return 0;
}
```

Listing 3.4: Example math expression represented in C code



```
[Info]
B1=b
B2=c
B3=2
B4=x
B5=y
B6=(B1+B2)     B6 mapped to res
B7=(B4+B5)     B7 mapped to res1
B8=(B6/B3)     B8 mapped to res
B9=(B8*B7)     B9 mapped to res
```

```
#include<stdio.h>
#include <memory.h>
#include<stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int aa,bb,cc,dd;
    int b =  0;
    int c =  0;
    int num0 = 2;
    int x =  0;
    int y =  0;
    int res = 0;
    int res1 = 0;
```

Figure 3.31: Variables and constants defined in obfuscated C code

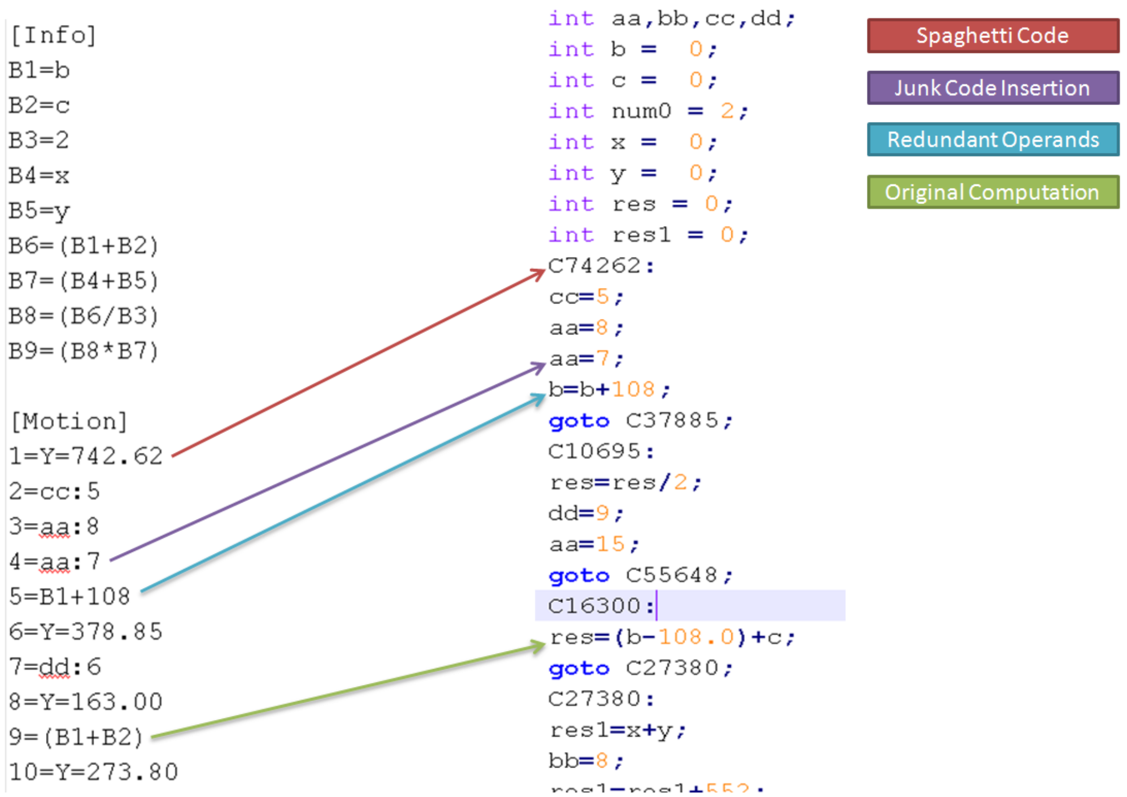
After updating the variable and constant definitions, the “Motion” section of the log file (Figure 3.30) is parsed to add the obfuscating transforms and original computations of the program. Each line of this section is read and parsed accordingly as outlined in Algorithm 2. Based on the line, the obfuscator script updates the C program either as a form of obfuscation such as junk code insertion, redundant operands, spaghetti code control statements, or as a relevant computation in the math expression.

Algorithm 2 Obfuscator

```
1: procedure OBFUSCATOR(logfile)
2:     Create an empty list codeblocks to hold list of blocks of code
       snippets for each GOTO block in the code
3:     Create an empty list currBlockCode to hold code snippets for a GOTO block
4:     Create an empty variable currGOTOBlock to keep track of the
       current GOTO block
5:     for each entry in “Motion” section of log file do
6:         Split the entry with delimiter “=”
7:         Store the second element after splitting entry in checkCode
8:         if checkCode contains “Y” then
9:             Remove decimal points from numerical value
10:            Create a new block name nextGOTOBlock by prepending “C” to the
               numerical value
11:            if nextGOTOBlock is the first GOTO block then
12:                currGOTOBlock = nextGOTOBlock
13:            else
14:                currGOTOBlock = nextGOTOBlock
15:                Update block code currBlockCode
16:                Map currGOTOBlock with currBlockCode and add to the
    codeblocks list
17:                Create a new empty list currBlockCode to hold code snippets for
                   the next GOTO block
18:                Set the nextGOTOBlock as currGOTOBlock
19:            end if
20:        else if checkCode contains “:” then
21:            Replace “:” with “=”
22:            Add the updated checkCode to currBlockCode list
23:        else
24:            checkCode represents a binary operation
25:            Parse checkCode into operands list operands and operator in operator
26:            Find and update operands with the code elements corresponding
               to identifier names in operands
27:            if checkCode represents a redundant expression then
28:                Update the redundant value with the operator and map it to the
                   code element in operands
29:                Convert checkCode to corresponding C code form
30:                Add the converted checkCode to currBlockCode list
31:            else
32:                checkCode represents actual computation in code
33:                Create updated expressions updateValue to update the original
                   value of code elements in operands from redundant values, if any
```

```
34:                 Replace the code elements with their $updateValue$ expression in
    $checkCode$
35:                 Convert $checkCode$ to corresponding C-code form
36:                 Add the converted $checkCode$ to $currBlockCode$ list
37:             end if
38:         end if
39:     end for
40: end procedure
```

The final updated C program computes the math expression. But the process of computation is obfuscated with various obfuscating transforms as shown in Figure 3.32.



```
[Info]
B1=b
B2=c
B3=2
B4=x
B5=y
B6=(B1+B2)
B7=(B4+B5)
B8=(B6/B3)
B9=(B8*B7)

[Motion]
1=Y=742.62
2=cc:5
3=aa:8
4=aa:7
5=B1+108
6=Y=378.85
7=dd:6
8=Y=163.00
9=(B1+B2)
10=Y=273.80
```

```
int aa,bb,cc,dd;
int b =  0;
int c =  0;
int num0 = 2;
int x =  0;
int y =  0;
int res = 0;
int res1 = 0;
C74262:
cc=5;
aa=8;
aa=7;
b=b+108;
goto C37885;
C10695:
res=res/2;
dd=9;
aa=15;
goto C55648;
C16300:
res=(b-108.0)+c;
goto C27380;
C27380:
res1=x+y;
bb=8;
```

Figure 3.32: Log file to C program mapping

3.2.6 Color Standards

This game relies on using colors to represent different code elements. What will happen if we have many code elements but few colors to represent them? This presents an interesting insight into the limitations of the code obfuscation process with this game. We tried to find out how many code elements could be mapped to distinct colors with the help of this game. Assuming the game will be played by a human player, he or she should be able to distinguish the colors used in the game. Our experiment was based on the process of computing a similarity measure between the colors used in the game. Using the similarity measure, we could apply a certain threshold value such that colors are perceptibly distinct to the human eye.

One of the metrics used to calculate the similarity between two colors is to measure the Euclidean distance between the colors using an appropriate color space reference. The color space has a specific mapping to a color model which provides a mathematical representation of colors [50]. For example, Adobe RGB [49] and sRGB [48] are related to the RGB (Red, Green, Blue) color model which expresses color in 3-tuples of red, green and blue. Two standard color spaces cover the range of colors which the normal human eye can perceive. These color spaces are CIELAB [51] and CIEXYZ [52]. Both of these color spaces are specified by the International Commission on Illumination, the international authority on color spaces. The CIEXYZ standard is based on X, Y and Z values which are extrapolations of R, G, B values to eliminate negative numbers and are known as tristimulus values [52]. The value Y is a measure of luminance which takes into account the response of the human eye to the "total power of light source" as given in the technical guide by Adobe on CIEXYZ [52]. However, a "non-uniform color scaling" issue was observed in CIEXYZ where colors with the same Euclidean distance appeared "farther apart" in the green color than red or blue in the color space. Then the next standard CIELAB was developed which we have adopted for our work in this research. The main aim of CIELAB was to become "device independent" whereas color models such as RGB were not [51]. CIELAB also showed an improvement over CIEXYZ with regards to "uniform color spacing" [51].

This color space is denoted with L, a, and b values. As described by the technical guide for CIELAB [51], L is a measure of lightness whereas a, and b represent colors.

We used the conversion formula of CIELAB from RGB values [54] to calculate the distance between the colors used for the game. We have used the CIE94 color difference formula [55] for our calculations. To maintain distinct color generation, we experimented with different thresholds starting from color difference threshold 1 and slowly incrementing it up to a threshold value of 28 as shown in Figure 3.33. We tried to find the maximum and minimum limits of the threshold that could be used to produce noticeably different colors. So we started with threshold 1, 2, 4 and then incremented the threshold value by 4 until we reached the maximum limit. To keep the experimental results limited and easy to understand, we incremented the threshold value by 4. Figure 3.33 (on page 55) shows the RGB values of colors plotted in three dimensions. As can be seen from the figure, with threshold 1, the colors are more concentrated. This means, at such a low threshold, we will get colors which are overlapping. Hence we will not get noticeably distinct colors applying this threshold when calculating distance between colors using the CIELAB metric. We may conclude that the quality of distinct color generation will reduce. With threshold 28, the colors are more spread apart. Hence the chances of generating noticeably distinct colors are greater. But due to the large distance threshold between the colors, there are fewer chances of finding *many* distinct colors. We may conclude that in such case, the quantity of distinct color generation will reduce. This was one trade-off we had to deal with during the course of our research. One of the options to decide on a threshold is to determine how many distinct colors we may need. Based on that, we could choose a lower threshold if we need more colors and compromise on the quality of distinct color generation. On the other hand, we could choose a higher threshold if we need fewer colors and maintain the quality of distinct color generation.

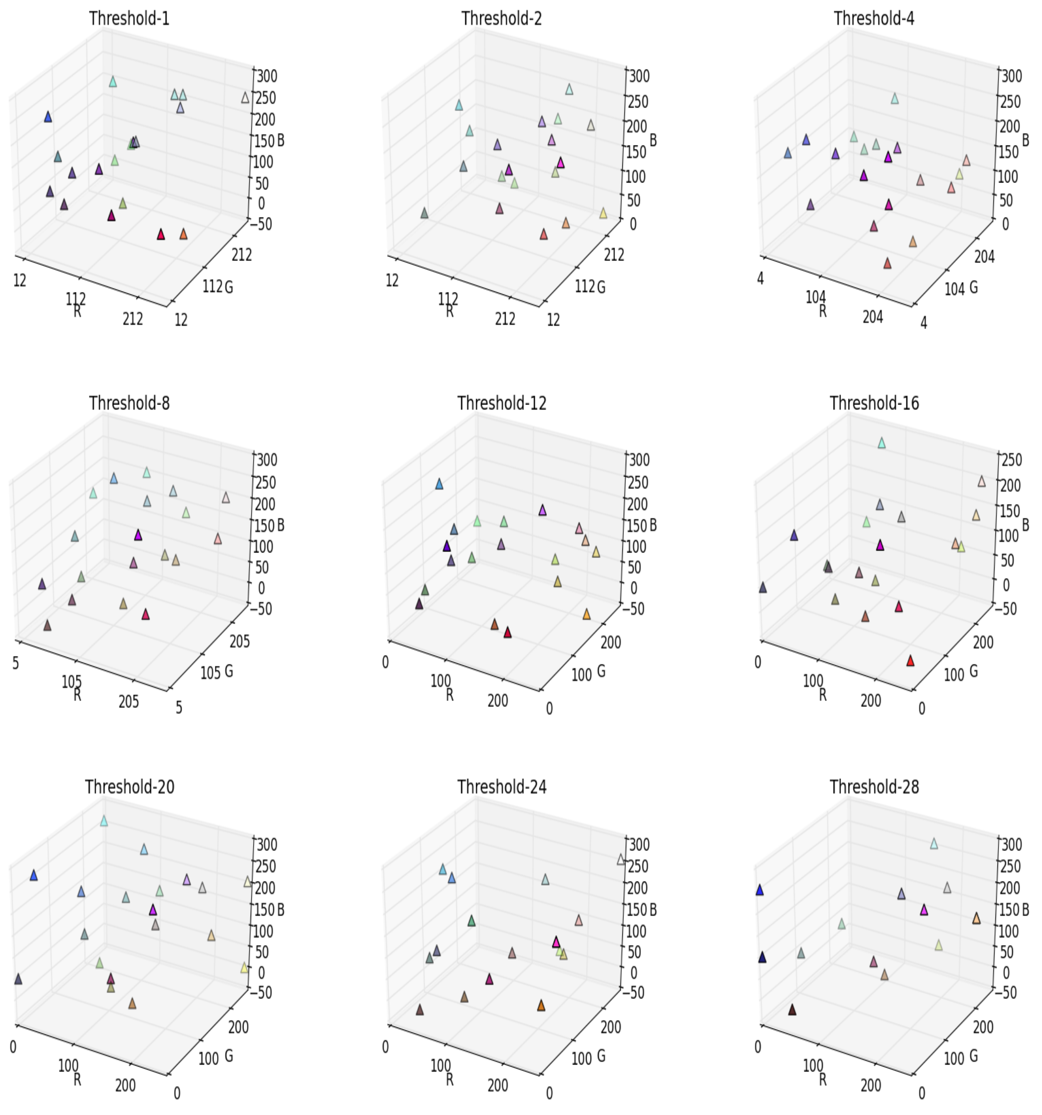


Figure 3.33: 3D plot of colors generated by different thresholds of color difference

Applying such color standards and threshold limits were useful to generate distinct colors for the game.

3.3 Evaluation of obfuscation output

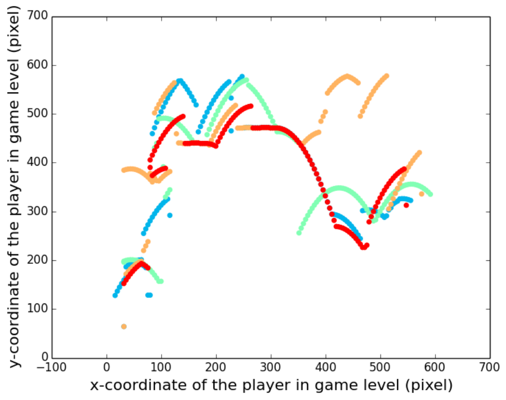


Figure 3.35: Player movement plots with static level played multiple times

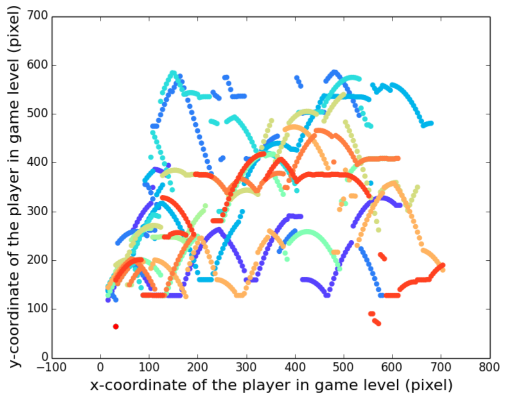


Figure 3.36: Player movement plots with dynamic PCG generated level played multiple times

We have been able to produce an obfuscated version of code for an arithmetic expression evaluation by playing the developed video game (Listing 3.5 (on page 57)). The values used for obfuscating transforms depend on the player movement in the game. Subsequently, player movement is controlled by the position of game elements in the game. Hence we have to vary the structure of each game level so that the output of the obfuscation varies each time the game level is played. If the game level has a static design, there will be minimal difference between the obfuscated output versions generated by playing the level multiple times as shown in Figure 3.35 (on page 56). This figure provides the plots of successful gameplay for a player playing the same static level four times. From this figure, it is evident that minimal changes in player movement could be detected while playing the static level many times. Therefore, we used a procedurally content generation engine to dynamically vary the game level structure. We were able to achieve noticeable randomness for different instances of same level as shown in Figure 3.36 (on page 56). To support our claim, we also plotted the player movement for different instances of the game level separately as shown in Figure 3.34 (page 60). These graphs record different player movements for different level instances. Hence we may conclude that using dynamic level generation process; the game can output different versions of obfuscated code.

```
#include <stdio.h>
#include <memory.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
 int aa,bb,cc,dd;
 int b =  0;
 int c =  0;
 int num0 = 2;
 int x =  0;
 int y =  0;
 int res = 0;
 int res1 = 0;
 C74262:
 cc=5;
 aa=8;
 aa=7;
 b=b+108;
 goto C37885;
 C10695:
 res=res/2;
 dd=9;
 aa=15;
 goto C55648;
 C16300:
 res=(b-108.0)+c;
 goto C27380;
 C27380:
 res1=x+y;
 bb=8;
 res1=res1+552;
 cc=7;
 goto C85138;
 C37885:
 dd=6;
 goto C16300;
 C48885:
 b=b-104.00;
 cc=892;
 b=b+140.00;
 bb=484;
 goto C55678;
 C55648:
 dd=191;
 b=b-132.00;
 y=y+324.00;
 dd=662;
 goto C69741;
 C55678:
 y=y+308.00;
 aa=518;
 bb=354;
 goto END;
 C69741:
 bb=486;
 aa=885;
 aa=872;
 bb=827;
 goto C48885;
 C85138:
 aa=237;
 cc=776;
 goto C10695;
 END:
 res=res*(res1-552.0);
 printf("res = %d",res);
return 0;
}
```

Listing 3.5: Obfuscated code

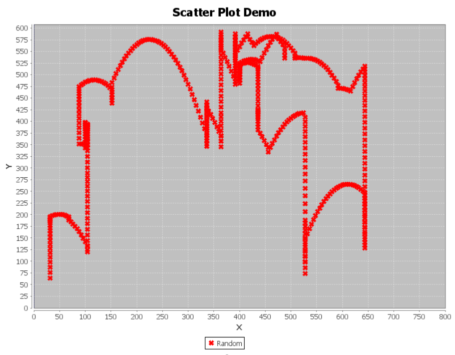
3.4 Summary

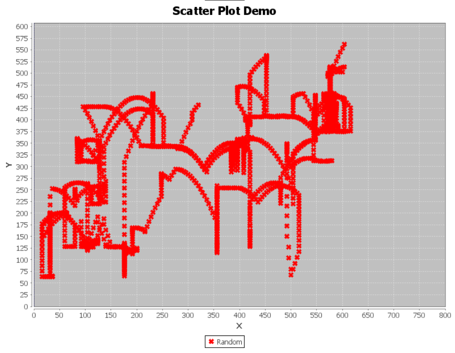
In this chapter, we demonstrated and analyzed how to obfuscate code by playing the research video game that we developed. The human player is the obfuscator in this technique. We decided to abstract the fact that the gaming actions by the player are used for obfuscated code generation. It helped us to be flexible while designing the game. This game uses colors for representing game elements and, indirectly, code elements of the code being obfuscated. The choice of colors is arbitrary and there could be other ways to design a game for code obfuscation purposes.

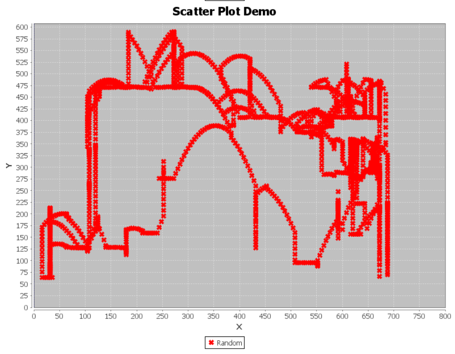
We presented an approach to map game elements to code elements. Player interactions with game elements are mapped to the operations in the code. For example, we used colored gems to represent the variables and constants in the code, and multicolored tiles with colored balloons to represent the binary operations in the code. Player avatar interactions with the colored balloons on multicolored tiles are mapped to operations in the code. Depending on these mappings, the gameplay is constrained so that the code in obfuscated version is generated in the same order as in the original code.

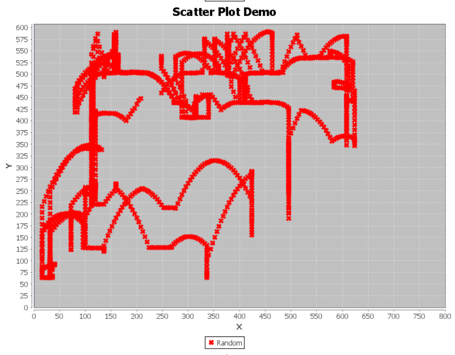
Some game elements were used to produce obfuscating transforms, such as neon tokens. The player avatar's vertical movement while bumping into the neon tokens is used for spaghetti code generation. The relative horizontal position of the player avatar on ladders produced by neon tokens is used for junk code generation. The player avatar's jumps on multicolored tiles produced the redundant operand obfuscation. Inducing multiple goals at a time in the game helped us achieve obfuscation by code re-ordering.

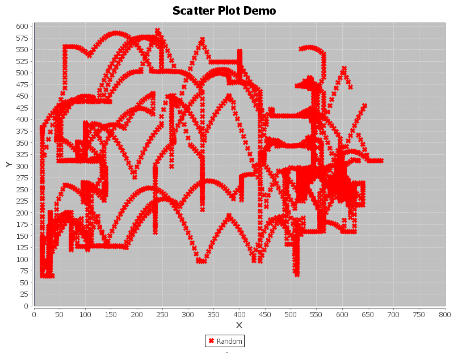
The bulk of obfuscating transforms generated by this game depend on player movement. We designed different versions of the game level using a procedurally content generation engine. This allowed us to create different obfuscated versions of the same code.

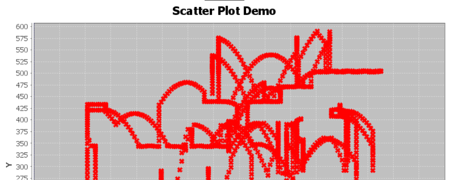

Scatter Plot Demo
Y
X
Random


Scatter Plot Demo
Y
X
Random


Scatter Plot Demo
Y
X
Random


Scatter Plot Demo
Y
X
Random


Scatter Plot Demo
Y
X
Random


Scatter Plot Demo
Y

Chapter 4

Code obfuscation through Super Mario

In the previous chapter, we discussed the game that we developed. The main objective for developing the game was to generate obfuscated code. Our efforts yielded the desired result with obfuscating transforms such as code re-ordering, junk code insertion and modifications with redundant operands. To follow this, we wanted to investigate if it is possible to modify an *existing* game to generate obfuscated code. If it is possible, then we may not need to invest efforts in developing a game for the same purpose.

How could we decide if an existing game has potential to generate code obfuscation? Our thoughts were focused on the ideas we used for the previous game that we developed for code obfuscation, a game which allows horizontal and vertical movement of the player. Therefore, we were looking for a suitable platform game which according to the definition [46] allows the player to jump on platforms and has enemies making the goal difficult to achieve. We also required a number of interactive game elements in the game such as enemies and rewards to be used as sources of randomness for the obfuscation. Another constraint was that we need to have access to the source code of the game, since we need to modify the game to extract data for code obfuscation.

We decided to use a version of Super Mario [57] which was used for the 2009 Mario AI competition. The game source code, including code for an AI agent player, was available via the Internet.

We identified the game elements which could be used to collect data for obfuscation, then the data was processed to generate the obfuscated code output. This game also used a built-in procedural content generation engine [44]. By setting certain parameters, it was possible to slightly vary the levels generated. This in turn helped us to generate different versions of obfuscated output.

With this game, we also had the option to compare the randomness generated by a human player and an AI agent.

This chapter is organized as follows: Section 4.1 presents the evolution of Super Mario to a code obfuscating game; Section 4.2 describes the technical implementation which processed Super Mario game output to produce the obfuscated code, followed by a discussion about the evaluation results of obfuscation and a brief summary in Sections 4.3 and 4.4.

4.1 Super Mario as a code obfuscating video game

The Mario AI competition is held by the IEEE Games Innovation Conference and the IEEE Symposium on Computational Intelligence and Games [57]. The motivation of this event is "to develop artificial intelligence (AI) for platform games" as described by the authors of this paper [57]. The game developed for the competition is based on Markus Persson's Infinite Mario Bros which is a copy of Nintendo's classic platform game Super Mario Bros. The Java source code of the game is available freely[1]. As discussed by the organizers of the competition in the paper [57], Infinite Mario Bros was re-factored to a Java API interface for the purpose of the competition. The participants develop an AI controller to play the game so that the controller can complete the game level with the highest possible score. The controller could be developed in any programming language provided they could be tested with the Java based competition software.

We had access to the source code of Markus Persson's Infinite Mario Bros[1] as well

[1]https://mojang.com/notch/mario/

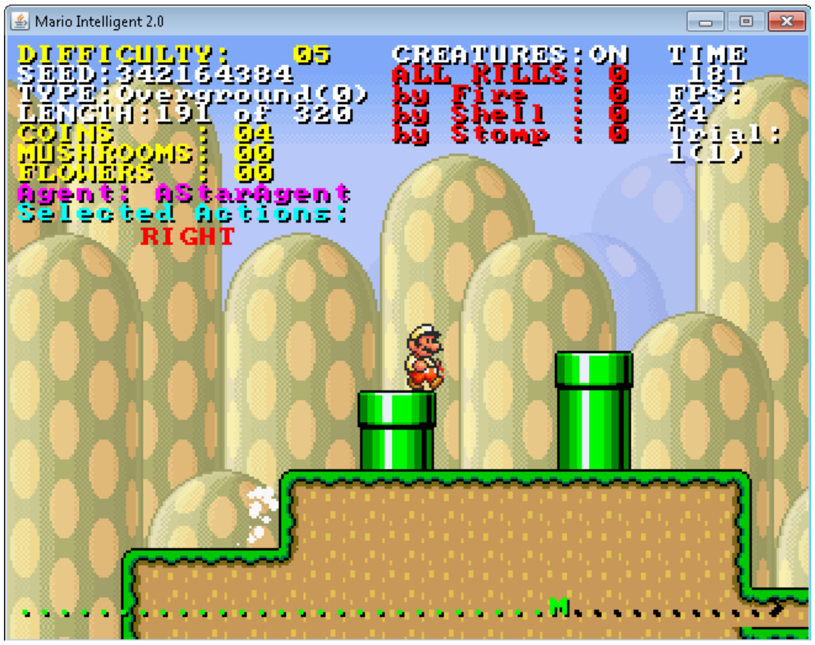


Figure 4.1: Super Mario - 2009 Mario AI Competition

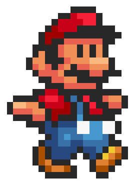
as the game software of the 2009 Mario AI competition[2]. Both are available publicly. Since AI controllers were used for the game competition, we wanted to test the ability of the AI controller to obfuscate code versus the human player. We had an intuition that a human player will generate more randomness than an AI controller. Hence we decided to select the competition game software and the winning AI controller of the 2009 Mario AI competition (developed by Robin Baumgarten) for our analysis.

The previous game we developed for code obfuscation was a platform game. So Super Mario was one of the probable choices as it also belongs to the genre of platform games. Since we have experience from the previous game and access to the source code of Super Mario, we had better control on how to derive obfuscating transforms from game elements used in the Super Mario game. A still from the game is shown in Figure 4.1.

[2]http://julian.togelius.com/mariocompetition2009/gettingstarted.php

4.1.1 Super Mario

Premise. The paper [57] describes the game-play and elements (Figure 4.2) of Super Mario in detail. The levels generated by this platform game are two-dimensional which presents a side-ways view for the player as evident from Figure 4.1. The main goal of the game is to complete the level by moving the player character, Mario, from left to right. Three factors influence the score in this game: the speed the level is completed; the number of coins collected by Mario; the number of enemies killed by Mario. While traversing the level, Mario has to survive the obstacles in the game such as holes in the level and moving enemies.


(a) Mario

(b) Enemies in order - Goomba, Spikey and Piranha Flowers

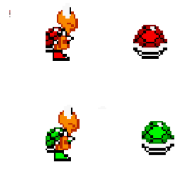

(c) Enemy turtles

Figure 4.2: Game elements from Super Mario

Game elements. The player character in the game is known as Mario. He can walk, run left or right, and can jump. There are three additional states which can decide what powers Mario can have - *small*, *big* and *fire*. In the big state, Mario can destroy objects by jumping on them from below. In the fire state, Mario can use fireballs as ammunition against enemies.

The enemies in the Super Mario game add the conflicts in the gameplay. The ground in the game is broken in places introducing gaps or holes. If Mario falls into any of the holes, he dies. The moving enemies in the game consist of turtle characters known as red koopas and green koopas. They get into a shell when stomped on by Mario. Then these turtles can be used by Mario to hit other enemies. Enemy characters such as goombas, spikey and cannon balls die when jumped on from above by Mario. Otherwise, if Mario bumps into them, he loses life and transitions into the small state. Enemies such as piranha plants are not affected by Mario at all but they are harmful for Mario. The best strategy is to avoid collision with such enemies.

The reward system in Super Mario consists of two elements, coins and mushrooms. Coins help in increasing the ultimate score in the game. The mushrooms act as power-ups which transition Mario to the big state. There are fire flowers which transition Mario into the fire state if Mario is already in the big state. When Mario is in the fire state, he gains the ability to shoot fireballs to kill enemies.

4.1.2 Game elements for obfuscating transforms

We were able to achieve obfuscating transforms such as junk code insertion, adding redundant operands to binary operations, spaghetti code obfuscating the control flow and code reordering using the Super Mario game. The same principles used for obtaining code obfuscations in the previous game worked for Super Mario as well, with some minor modifications. The data used for the obfuscation is recorded at regular time intervals while the player is playing the Super Mario game. The data logged consists of the difference in the x and y-coordinate values of Mario, the current time in seconds, the number of coins collected, the number of lives available, and the number of creatures killed at that point in the game.

The difference in the y-coordinate of Mario's position was used for labels for generating spaghetti code flow. The time value in seconds was used for junk code insertion. All values collected at a single time interval were added and used for adding redundant operand obfuscation. In the previous game developed by us, we had the flexibility to introduce multiple goals while designing the level. For example, we could provide different orders to pop balloons to advance in the game. This flexibility helped us to achieve code re-ordering obfuscation from player inputs. But for Super Mario, the only goal for the player is to complete the level. We did not have the flexibility to design game levels as it was generated by the automated level generator used in the Super Mario game. Hence we could not obtain player inputs to achieve code re-ordering obfuscation. Code re-ordering was achieved by internally shuffling the code while generating the obfuscated code output. The section on technical implementation of obfuscated code provides more detailed description of the obfuscation process.

Level Generation in Mario. One of the important features of the Super Mario game used for the 2009 Mario AI competition is automatic level generation as described in the paper [57]. While generating the fixed-width level for the game, game elements such as blocks, holes, and opponents are added based on defined parameters. The level generator uses a random seed value to specify the difficulty of the level, and the number and positions of the game elements in the generated level. In this way, any particular level could be re-generated using the same seed value.

4.2 Technical implementation of code obfuscation with Super Mario

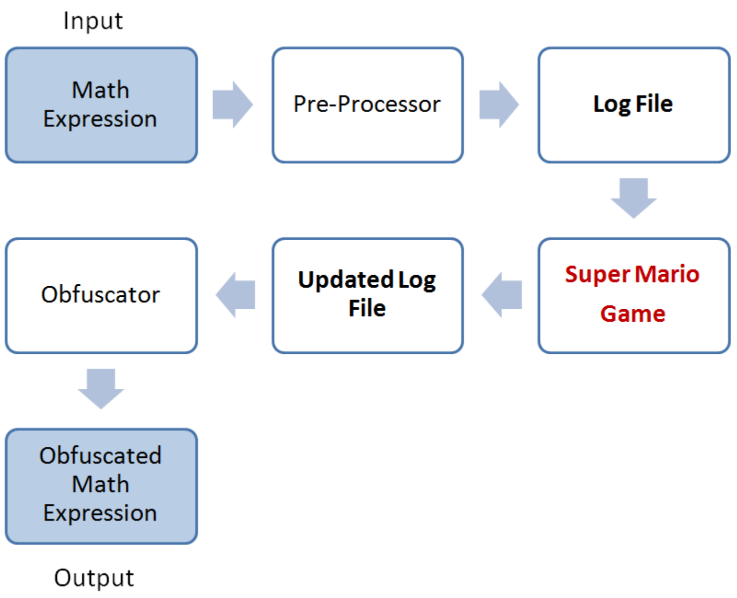


Figure 4.3: The process flow of code obfuscation using Super Mario

```
#include <stdio.h>
int main()
{
 int b, c, x, y, res, res1;
 res = b + c;
 res = res / 2;                  //res = (b+c)/2
 res1 = x + y;
 res = res * res1;                //res = ((b+c)/2) * (x+y)
 printf("Calculated value = %d\n",res);
 return 0;
}
```

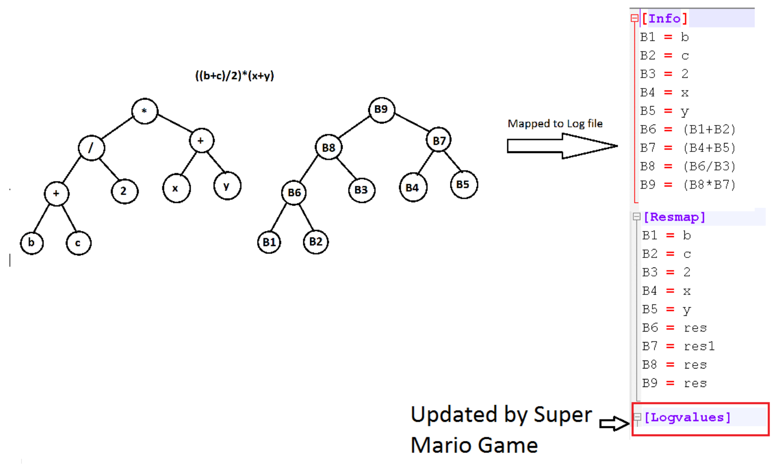
Listing 4.1: Example arithmetic expression in C code

In this section, we will explain the technical details about generating obfuscated code using Super Mario. The original code (Listing 4.1) is used to explain the obfuscation process which computes the arithmetic expression $((b+c)/2)*(x+y)$. To maintain consistency, we have used the same math expression for the obfuscation as used for the previous game. A flow of the obfuscation process is shown in Figure 4.3. The process flow is similar to the one we used for our previous game, except that the log file generated by the pre-processor is used only to log data from Super Mario (in the previous game, the log file was used to design levels of the game as well). The pre-processor accepts an arithmetic expression as input and generates a log file. The information in the log file is later used by the obfuscator to re-generate the original code flow in the obfuscated output. The log file is updated with player inputs as discussed before when the player is playing the game level. Then the obfuscator processes the player inputs as well as the code flow information updated by the pre-processor to generate the final obfuscated output. The segments of the process flow (Figure 4.3) are explained in detail in the following sections.

4.2.1 Pre-processor

The pre-processor is a Python script which parses the arithmetic expression and creates the initial version of the log file in the process. It updates the file with the variables and constants used in the math expression as well as the order of execution of the operations.

Following the same mechanism as used for the previous game, the expression $((b + c)/2) * (x + y)$ is grouped into binary operations as evident from Listing 4.1. The log file generated by the pre-processor is shown in Figure 4.4. The “Logvalues” section in the file is updated with player input when the game is being played.



```
[Info]
B1 = b
B2 = c
B3 = 2
B4 = x
B5 = y
B6 = (B1+B2)
B7 = (B4+B5)
B8 = (B6/B3)
B9 = (B8*B7)

[Resmap]
B1 = b
B2 = c
B3 = 2
B4 = x
B5 = y
B6 = res
B7 = res1
B8 = res
B9 = res

[Logvalues]
```

Figure 4.4: Log file generated by the pre-processor

The expression is arranged in the form of a sequence of binary operations and represented by an abstract syntax tree (AST) [45] as shown in Figure 4.4. Each node of the tree is labeled with a number with the prefix “B”. The pre-processor script is the same as used for the previous game. The sections added in the log file are “Info”,“Resmap”, and “Logvalues”.

4.2.2 Super Mario Game - Updating the log file

We modified the Java source code of the Super Mario game so that while the game is being played, the player inputs could be logged in the log file. The main Java program to start the game is “Play.java” in the package “ch.idsia.scenarios”. It sets the agent who is going to play the game. There could be two types of agents - a HumanKeyboardAgent played by a human player and an artificially intelligent (AI) agent. It also defines several options such as the random seed used to generate the game level and the difficulty level of the game. The class “SimulationOptions” in the package “ch.idsia.mario.simulation” is used to set the simulation options for the game. As discussed, we need to log game statistics such as the number of coins earned and the number of enemies killed at regular time intervals. So we analyzed the source code of Super Mario and found the class “LevelScene” in “ch.idsia.mario.engine” package which renders the game information on-screen.

We developed a new class “LoggingTask” in the “ch.idsia.ai.tasks” package which updates the player inputs in the log file. It computes the difference in x and y coordinates of Mario in the game and extracts the game information from the “Mario” class instance and the “LevelScene” class instance maintained during the game session. Another method of this class is to compute and update the MD5 hash [58] of x and y coordinate differences. This MD5 data is later used to compare with the original data generated by the game for randomness.

We updated the “SimulationOptions” class to include a new instance of “LoggingTask” each time the game is played. We modified the “LevelScene” class to invoke the logging method of “LoggingTask” class instance after every second.

4.2.3 Input log file for obfuscation

After the game is played, the log file is updated as shown in Figure 4.5 (page 71). The symbol “:” is used as the delimiter for the values logged. The first line of the “Logvalues” section in Figure 4.5 gives the order in which the values are logged. The order is

“DelX:DelY:timeInSec:coins:lives:killedCreaturesTotal”

The “DelX” term means the difference between the previous and current x-coordinate of the player character, and “DelY” means the same for the y-coordinate. The “coin” term means the number of coins collected by Mario, the “lives” term means the number of lives available to the player, and the “killedCreaturesTotal” term means the total number of enemies killed by Mario when the data was being logged.

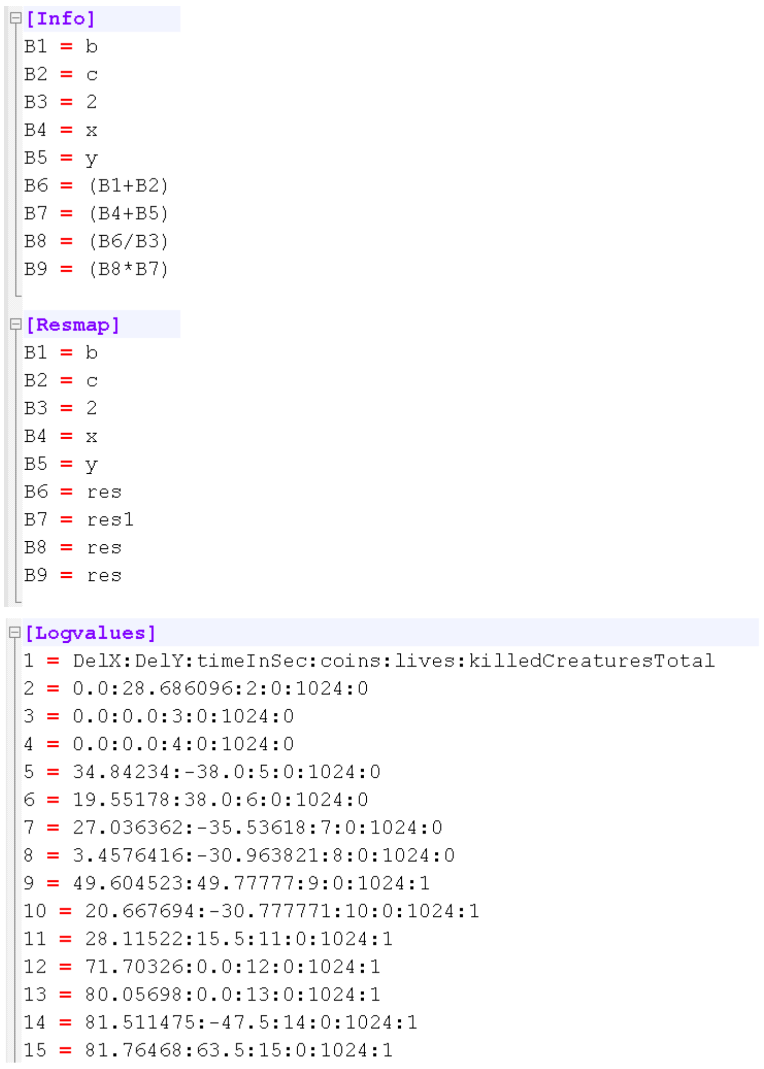

```
[Info]
B1 = b
B2 = c
B3 = 2
B4 = x
B5 = y
B6 = (B1+B2)
B7 = (B4+B5)
B8 = (B6/B3)
B9 = (B8*B7)

[Resmap]
B1 = b
B2 = c
B3 = 2
B4 = x
B5 = y
B6 = res
B7 = res1
B8 = res
B9 = res

[Logvalues]
1 = DelX:DelY:timeInSec:coins:lives:killedCreaturesTotal
2 = 0.0:28.686096:2:0:1024:0
3 = 0.0:0.0:3:0:1024:0
4 = 0.0:0.0:4:0:1024:0
5 = 34.84234:-38.0:5:0:1024:0
6 = 19.55178:38.0:6:0:1024:0
7 = 27.036362:-35.53618:7:0:1024:0
8 = 3.4576416:-30.963821:8:0:1024:0
9 = 49.604523:49.77777:9:0:1024:1
10 = 20.667694:-30.777771:10:0:1024:1
11 = 28.11522:15.5:11:0:1024:1
12 = 71.70326:0.0:12:0:1024:1
13 = 80.05698:0.0:13:0:1024:1
14 = 81.511475:-47.5:14:0:1024:1
15 = 81.76468:63.5:15:0:1024:1
```

Figure 4.5: Values from Super Mario logged in the log file

4.2.4 Obfuscator

The obfuscator is a collection of Java files which parses the log file (Figure 4.5) and generates the obfuscated output. Just like the previous game, the obfuscated output is a C program which follows the pattern set by a standard C template. The obfuscator script updates the C program with the processed obfuscated transforms.

The process of generating the obfuscated file is same as the previous game. First a new file named as "obfuscatedcode.c" is created and updated with the content of the template C file.

At first the variables are updated in the new C file using the data from the "Info" and "Resmap" section of the log file as shown in Figure 4.5. The algorithm for variable definition is given in Algorithm 3.

Algorithm 3 Update variable initializations in C file

```
1: procedure VarInitializations(logfile)
2:     Define a number count numcnt to use for variable definitions for numbers
3:     Iterate over the key-value pairs (key,value) of "Info" section of logfile
4:     if value is an expression then
5:         Refer to "Resmap" section of logfile for variable name
6:         Update the variable definition in C file
7:     else if value is a number then
8:         Increment numcnt, append to "num" as variable name
9:         Update the variable definition in C file
10:        Update the variable name in "Resmap" section of logfile for future reference
11:    else
12:        Update the value as variable in C file
13:    end if
14: end procedure
```

Next the "Logvalues" section is parsed. The data for obfuscation is extracted from each line of the "Logvalues" section and corresponding data obfuscations are generated using Algorithm 4. One different method used here for generating obfuscation is to use the static single assignment (SSA) form of the program [59]. This technique was not used for the previous game. In SSA form, each variable is assigned once. Every consecutive update of a variable involves assignments with a new variable name as shown in Figure 4.7. This makes it easier to update and manipulate the variables in the program with redundant expression obfuscations and code reordering. The listing in Figure 4.6 (page 73) shows that in straight-line code, it is not possible to swap s4 before s3. But for the single static assignment form of the same code, s4 could be reordered before s3.

```
s1: c=2
s2: x=10
s3: a=b+c
s4: c=x+7
s5: d=c-9
```

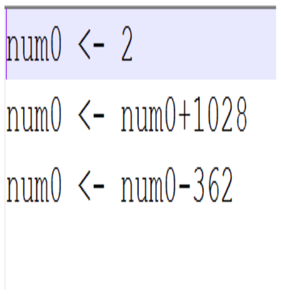
Listing 4.2: Straight-line code

```
s1: c1=2
s2: x1=10
s3: a1=b1+c1
s4: c2=x1+7
s5: d=c2-9
```

Listing 4.3: Single static assignment form

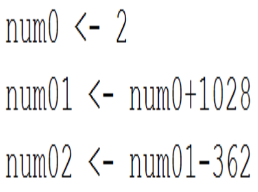
Figure 4.6: Code reordering using single static assignment form

For junk code insertion and redundant expression obfuscation, we had to randomly select variable names. The main idea was to use the randomness produced by the gameplay. We generated the binary bit-stream of XOR values (relative distance of Mario, x and y coordinate, and absolute time value, XORed). Our analysis regarding the randomness of the bit-stream is discussed later. We read the required number of bits from this bit-stream to determine the random variable to use for obfuscation.



```
num0 <- 2
num0 <- num0+1028
num0 <- num0-362
```

(a) Code before SSA



```
num0 <- 2
num01 <- num0+1028
num02 <- num01-362
```

(b) Code after SSA

Figure 4.7: Variable assignment with SSA

Algorithm 4 Update code obfuscations in C file

```
1: procedure GenObfuscations(logfile, binary bit stream)
2:     GOTOLabel list to archive the labels $GOTOLabels$ from the logged data
3:     List to store all variables updated with redundant expressions
       $redundantValObfuscationList$
4:     for each $line$ in “Logvalues” section of log file do
5:         Split the $line$ with delimiter “:” and extract the data as $datalist$
6:         Extract $DelY$ value from $datalist$ and add to $GOTOLabels$
7:         Select a random junk variable name $junkvar$ from list of junk
           variable names. The random junk variable is determined by read-
           ing bits from binary bit stream
8:         Extract $timeInSec$ from $datalist$
9:         Using $junkvar$ and $timeInSec$, generate junk code $junkcode$ for example
   $dd = 7.0$
10:        Update $junkcode$ in the C file
11:        Select a random operator $op$ from operator list $+, -, *, /$ by read-
           ing the random bits from binary bit stream
12:        Select one of the variables $varToUpdate$ randomly from variables
           already defined in the C file. The selection is determined by bits
           from binary bit stream.
13:        Compute the total of all values $totVal$ stored in $datalist$
14:        if $varToUpdate$ is in the $redundantValObfuscationList$ then
15:            Get the previous redundant value for $varToUpdate$ from
               $redundantValObfuscationList$
16:            Update the value of $varToUpdate$ with the new value $totVal$
17:            Update the new $varToUpdate$ in $redundantValObfuscationList$
18:        end if
19:        Get a new SSA name $varToUpdateSSAName$ for $varToUpdate$
20:        Update variable initializations in C file
21:        Update $varRedundantExprsn$ with
           $varToUpdateSSAName$, “=”, $varToUpdateExprsn$, $op$ and
           $totVal$
22:        Update $varRedundantExprsn$ in the C file
23:    end for
24: end procedure
```

After the obfuscating transforms are updated in the C file, the actual code flow is appended at the end of the file using the “Info” and “Resmap” section of the logfile. Then the actual code is shuffled in the program maintaining the control dependency of static single assignment. Next a random line of code (decided by reading a fixed number of bits from the binary bit stream) is picked from the updated C program and shuffled either up or down in the code. Algorithm 5 shows the code shuffle up in the C program.

Algorithm 5 Shuffle code in C program

```
1: procedure SHUFFLE CODE(C program)
2:     Initialize the number of lines of code to shuffle as $noOfShuffles$
3:     while $noOfShuffles$ is not zero do
4:         Select a random line of code to shuffle as $shuffleCode$
5:         while $shuffleCode$ does not have an assignment operation do
6:             Select a random line of code to shuffle as $shuffleCode$
7:         end while
8:         Extract and store the right hand side variables of the assignment
           operation in $shuffleCode$ as $RHSVars$
9:         Define the $shuffleCodeLineNo$ to set the line number of
           $shuffleCode$ in C program
10:        for each line of code code starting from $shuffleCodeLineNo$-1 until the
   first line
               in C program do
11:            if code has “=” operator then
12:                Extract the left hand side variable $LHSVar$ from code
13:                if $LHSVar$ matches any of the $RHSVars$ then
14:                    Move $shuffleCode$ in the next line after code in C program
15:                    break
16:                end if
17:            end if
18:        end for
19:    end while
20: end procedure
```

The “GOTO” labels are applied after the code shuffling is done. While parsing the “Logvalues” section of the logfile, “DelY” values are stored as labels. Next the code in the C program is divided into chunks of code. The number of lines of code in each chunk is decided randomly using the bits from the binary stream. But that number is restricted using the “mod” function. The chunks of code are then assigned labels such as “L4212906:” and next labels to go to are appended at the end of the chunk of code as “goto L58600006;”. Then the chunks of code are sorted in ascending order based on the labels. The final output is the sorted labeled C program.

4.2.5 Output obfuscated C code

The original code in Listing 3.4 (page 50) is obfuscated by playing Super Mario. The obfuscated output is given in Listing 4.4. A longer version of the obfuscated output from Super Mario with data logged from multiple rounds of play is given in Appendix B.

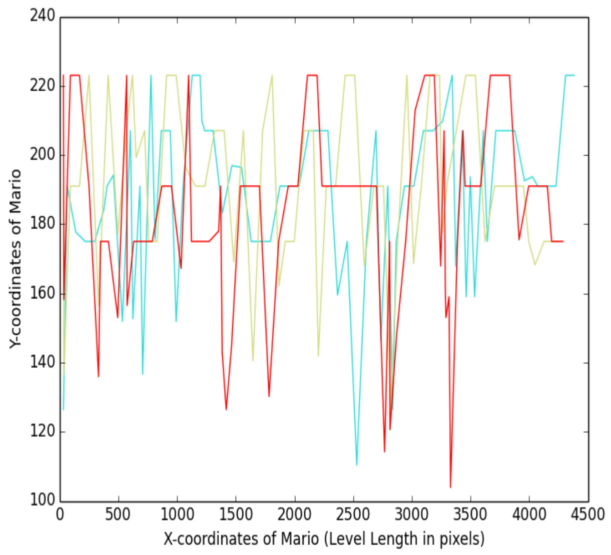
The obfuscated output from Super Mario is longer as compared to the obfuscated output from the previous game. Using SSA for variable assignments may be one of the reasons for the lengthy code. Subscripts were used to rename variables each time they are re-assigned. For example, in case of variable "y", it is renamed with "y0","y_1","y_2" through "y_14". The same process is followed for other variables as well. The threshold limit for lines of code in a "GOTO" block was set to be 9. Hence the lines of code in each "GOTO" blocks vary from 1 to 9.

Listing 4.4: Obfuscated code with Super Mario

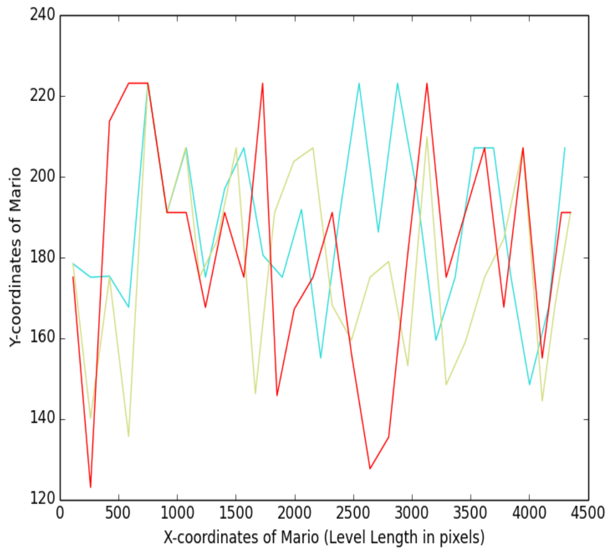
```
#include <stdio.h>
#include <memory.h
   >
#include <stdlib.h
   >
#include <string.h
   >

int main(int argc,
    char *argv[])
{
 int aa,bb,cc,dd;
 int b =  0;
 int c =  0;
 int num0 = 2;
 int x =  0;
 int y =  0;
 int res = 0;
 int res1 = 0;
 int b0 = 0;
 int res00 = 0;
 int num00 = 0;
 int res_01 = 0;
 int res10 = 0;
 int res_02 = 0;
 int res_03 = 0;
 int res_04 = 0;
 int res_05 = 0;
 int res_11 = 0;
 int b_1 = 0;
 int y0 = 0;
 int c0 = 0;
 int num_1 = 0;
 int c_1 = 0;
 int res_12 = 0;
 int res_06 = 0;
 int num_2 = 0;
 int res_07 = 0;
 int res_13 = 0;
 int res_14 = 0;
 int num_3 = 0;
 int res_08 = 0;
 int num_4 = 0;
 int res_09 = 0;
 int x0 = 0;
 int num_5 = 0;
 int res_15 = 0;
 int res_010 = 0;
 int res_011 = 0;
 int b_2 = 0;
 int c_2 = 0;
 int num_6 = 0;
 int x_1 = 0;
 int y_1 = 0;
 int res_012 = 0;
 int res_16 = 0;
L13300003:
 cc = 2.0;
 aa = 14.0;
 b0 = b+1051.0;
 aa = 3.0;
 res00 = res
    +1039.0;
 dd = 4.0;
 num00 = num0
    +1040.0;
goto L00;
L00:
 cc = 5.0;
 res_01 = res00
    -1045.0;
 bb = 6.0;
goto L10;
L10:
 res10 = res1
    -1047.0;
 cc = 7.0;
 res_02 = res_01
    -1052.0;
 cc = 8.0;
 res_03 = res_02
    -1061.0;
 cc = 9.0;
 res_04 = res_03
    -1059.0;
goto L31243591;
L20:
 b_2 = b_1+14.0;
 bb = 13.0;
 y0 = y+1070.0;
goto L8304993;
L30:
 num_1 = num00
    +1074.0;
 cc = 16.0;
 c_1 = c0+1082.0;
 c_2 = c_1
    -2149.0;
goto L40;
L40:
 aa = 17.0;
 res_12 = res_11
    -1078.0;
 bb = 18.0;
 res_06 = res_05
    -1077.0;
goto L50;
L50:
 bb = 19.0;
 num_2 = num_1
    +1078.0;
 res_08 = res_07
    -1083.0;
 bb = 20.0;
 res_07 = res_06
    +1079.0;
 cc = 21.0;
 res_13 = res_12
    -1080.0;
goto L60;
L60:
 dd = 22.0;
 res_14 = res_13
    -1081.0;
 aa = 23.0;
 num_3 = num_2
    +1088.0;
 bb = 24.0;
 cc = 25.0;
 num_4 = num_3
    +1084.0;
 cc = 26.0;
goto L405925;
L70:
 res_15 = res_14
    -1077.0;
 res_16 = res_15
    +6427.0;
 res_16=x_1+y_1;
bb = 30.0;
 res_010 = res_09
    -1078.0;
 dd = 31.0;
 res_011 =
    res_010
    -1079.0;
 res_012 =
    res_011
    +6394.0;
goto END;
 L405925:
  res_09 = res_08
     +1085.0;
  aa = 27.0;
  x0 = x+1086.0;
  x_1 = x0-1086.0;
  cc = 28.0;
  num_5 = num_4
     +1076.0;
  num_6 = num_5
     -6440.0;
  bb = 29.0;
 goto L70;
 L8304993:
  y_1 = y0-1070.0;
  c0 = c+1067.0;
  aa = 15.0;
 goto L30;
 L31243591:
  bb = 10.0;
  res_05 = res_04
     -1063.0;
  cc = 11.0;
 goto L0035751343;
 L0035751343:
  res_11 = res10
     -1064.0;
  bb = 12.0;
  b_1 = b0-1065.0;
 goto L20;
 END:
  res_012=b_2+c_2;
  res_012=res_012/
     num_6;
  res_012=res_012*
     res_16;
 printf("%d \n",
    res_012);
return 0;
}
```

4.3 Evaluation of obfuscation results

We were able to generate code obfuscation by playing Super Mario as evident from the obfuscated output discussed in the previous section. The obfuscations used in the code were derived from player movement and interaction with game elements. The game levels were developed by a built-in procedural content generator with a random seed value and a set difficulty level. The difficulty level for Super Mario used for all experiments in this obfuscation was set to 5. If the same seed is used, it is possible to generate the same level. Our main goal was to produce an obfuscated version of the original code and to explore ways to generate different obfuscated versions. One way is to vary the level design which will influence player movement in the game. Hence different obfuscated versions will be produced. We recorded and plotted the motion path of both human player and AI agent playing the same game level multiple times keeping the difficulty level fixed. The plots are shown in Figure 4.8. It is evident from the figure that not much variance is achieved even though the levels are randomly generated with a random seed, so we tried several ways to generate more randomness based on the movement of the human player in the game.



(a) Human Player



(b) AI agent

Figure 4.8: Plot of human player and AI agent playing the same level multiple times in Super Mario

4.3.1 Measure of Randomness

We had discussed the characteristics of random and pseudo-random number generators in the "Related Work" chapter of this thesis. One way to test such generators for randomness is to use certain statistical methods. Such techniques were assessed, utilized and combined in a test suite developed by the National Institute of Standards and Technology (NIST) [60]. These tests evaluate the "probabilistic" property of a random sequence. In other words, these tests measure the unpredictability of the output of a random sequence.

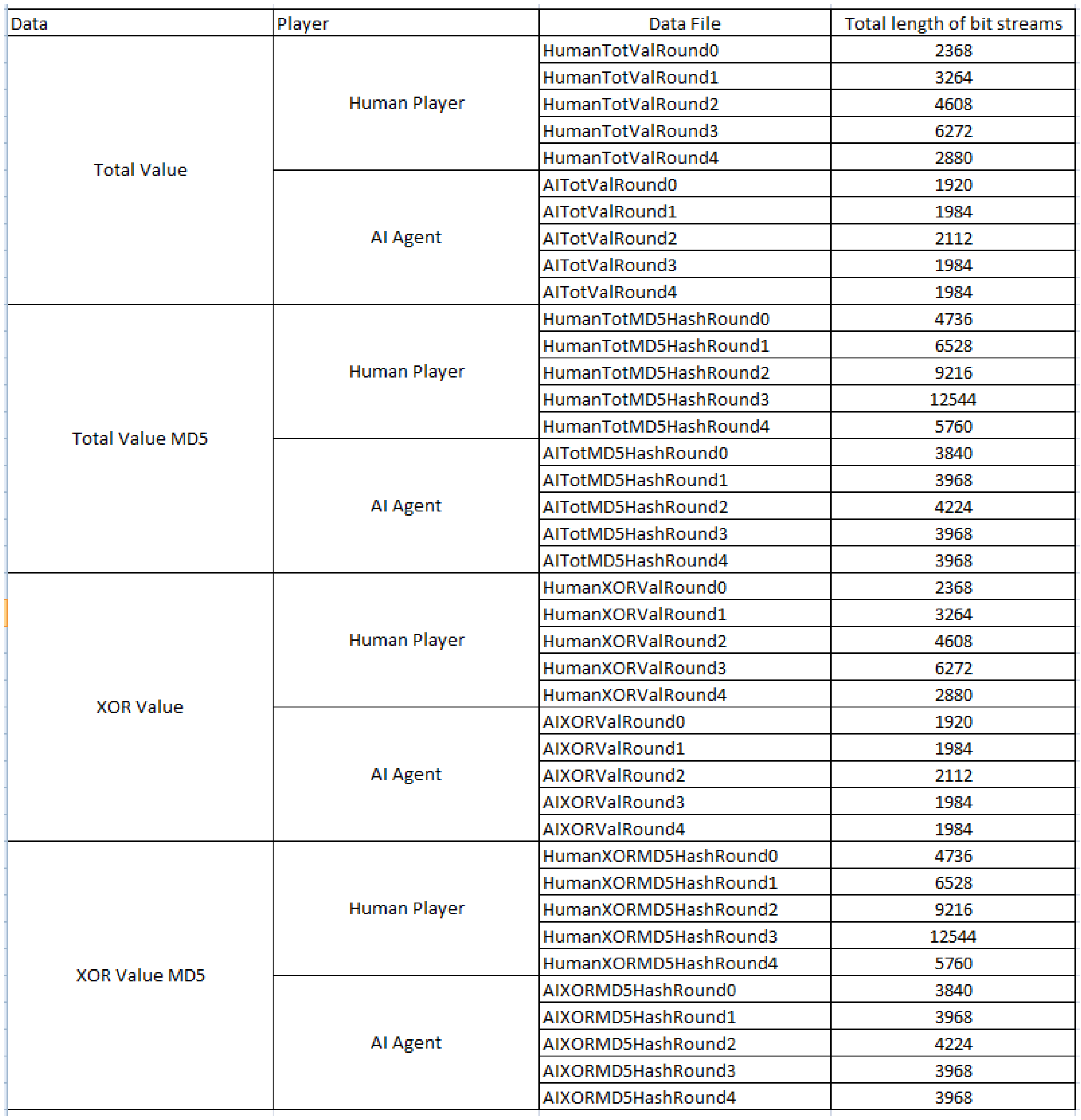
The test suite consists of 15 statistical tests. Each test requires a binary random sequence of 1's and 0's as input for testing. There is a minimum threshold limit for the length of the bit stream for each test to provide accurate results. Some tests such as the Frequency (Monobit) Test operate on the entire bit stream to determine the test results. Some tests such as the Block Frequency Test divide the bit stream into blocks of bits and then conduct the test. The description of the tests, recommendations of the input bit stream length and the pass condition is provided in the NIST paper [60].

We did not use all tests available in the NIST test suite due to size constraints of the bit stream. The minimum length of bit streams required for the test results to be valid is mentioned in the "Input Size Recommendation" section of randomness test descriptions in the NIST paper [60]. We were able to successfully use 8 out of 15 tests of NIST test suite. The tests we used are: Frequency (Monobit) Test, Block Frequency Test, Runs Test, Longest Run of Ones in a Block, Non-overlapping Template Matching Test, Approximate Entropy Test, Cumulative Sums (Cusum) Test, and Serial Test. The maximum length of bit stream we used for testing was slightly above 12000 bits.

Analysis of random testing results. We collected raw data for our randomness analysis from playing several rounds of the Super Mario game by both a human player and by the AI agent which won the 2009 Mario AI competition [57]. Each "player" played five rounds of the game.

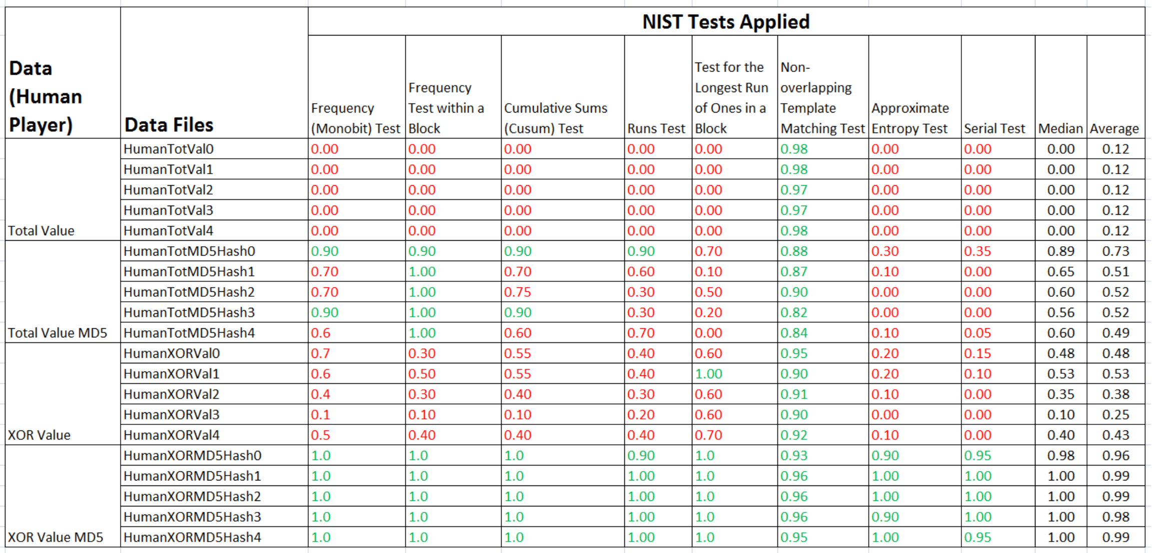
The raw data collected consists of the relative distance of x (delta-x) and y (delta-y) coordinates of Mario in the game, the time values in seconds, and the total value of game elements (number of enemies killed, number of lives, and number of coins collected by Mario). The data was logged after a set time interval which was 1 second for this sample.

The exclusive-or function is applied to delta-x, delta-y and time values to compute their XOR value. Then the corresponding MD5 hash of the XOR value is computed. The values are converted to binary bit-streams. Similarly the total value of game elements and corresponding MD5 value of the total value is computed and converted to binary bit streams. Hence for five rounds of play by human player, we had 5 bit streams of XOR values, 5 bit streams of MD5 hashes of XOR values, 5 bit streams of total values of game elements and 5 bit streams of MD5 hashes of total values. The same number of bit streams was also collected for five rounds of play by the AI agent. Therefore, we had 20 bit streams of binary data each, to test for randomness of the human player as well as the AI agent. The length of bit stream generated for each round of play is given in Figure 4.9 (page 82).



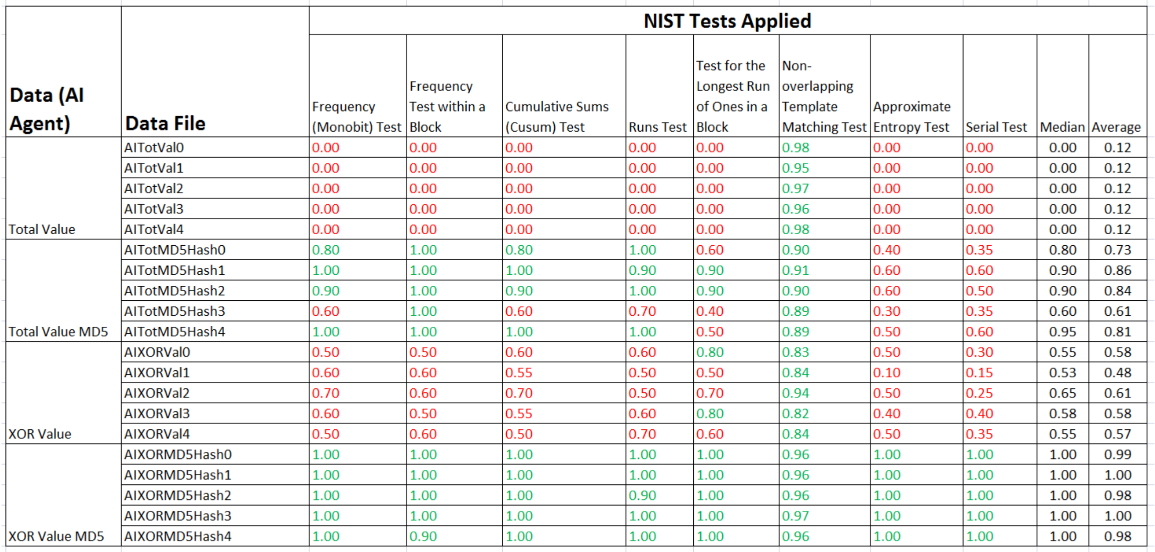
Data	Player	Data File	Total length of bit streams
Total Value	Human Player	HumanTotValRound0	2368
		HumanTotValRound1	3264
		HumanTotValRound2	4608
		HumanTotValRound3	6272
		HumanTotValRound4	2880
	AI Agent	AITotValRound0	1920
		AITotValRound1	1984
		AITotValRound2	2112
		AITotValRound3	1984
		AITotValRound4	1984
Total Value MD5	Human Player	HumanTotMD5HashRound0	4736
		HumanTotMD5HashRound1	6528
		HumanTotMD5HashRound2	9216
		HumanTotMD5HashRound3	12544
		HumanTotMD5HashRound4	5760
	AI Agent	AITotMD5HashRound0	3840
		AITotMD5HashRound1	3968
		AITotMD5HashRound2	4224
		AITotMD5HashRound3	3968
		AITotMD5HashRound4	3968
XOR Value	Human Player	HumanXORValRound0	2368
		HumanXORValRound1	3264
		HumanXORValRound2	4608
		HumanXORValRound3	6272
		HumanXORValRound4	2880
	AI Agent	AIXORValRound0	1920
		AIXORValRound1	1984
		AIXORValRound2	2112
		AIXORValRound3	1984
		AIXORValRound4	1984
XOR Value MD5	Human Player	HumanXORMD5HashRound0	4736
		HumanXORMD5HashRound1	6528
		HumanXORMD5HashRound2	9216
		HumanXORMD5HashRound3	12544
		HumanXORMD5HashRound4	5760
	AI Agent	AIXORMD5HashRound0	3840
		AIXORMD5HashRound1	3968
		AIXORMD5HashRound2	4224
		AIXORMD5HashRound3	3968
		AIXORMD5HashRound4	3968

Figure 4.9: Length of binary bit streams collected for NIST tests



Data (Human Player)	Data Files	NIST Tests Applied									
		Frequency (Monobit) Test	Frequency Test within a Block	Cumulative Sums (Cusum) Test	Runs Test	Test for the Longest Run of Ones in a Block	Non-overlapping Template Matching Test	Approximate Entropy Test	Serial Test	Median	Average
	HumanTotVal0	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.12
	HumanTotVal1	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.12
	HumanTotVal2	0.00	0.00	0.00	0.00	0.00	0.97	0.00	0.00	0.00	0.12
	HumanTotVal3	0.00	0.00	0.00	0.00	0.00	0.97	0.00	0.00	0.00	0.12
Total Value	HumanTotVal4	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.12
	HumanTotMD5Hash0	0.90	0.90	0.90	0.90	0.70	0.88	0.30	0.35	0.89	0.73
	HumanTotMD5Hash1	0.70	1.00	0.70	0.60	0.10	0.87	0.10	0.00	0.65	0.51
	HumanTotMD5Hash2	0.70	1.00	0.75	0.30	0.50	0.90	0.00	0.00	0.60	0.52
	HumanTotMD5Hash3	0.90	1.00	0.90	0.30	0.20	0.82	0.00	0.00	0.56	0.52
Total Value MD5	HumanTotMD5Hash4	0.6	1.00	0.60	0.70	0.00	0.84	0.10	0.05	0.60	0.49
	HumanXORVal0	0.7	0.30	0.55	0.40	0.60	0.95	0.20	0.15	0.48	0.48
	HumanXORVal1	0.6	0.50	0.55	0.40	1.00	0.90	0.20	0.10	0.53	0.53
	HumanXORVal2	0.4	0.30	0.40	0.30	0.60	0.91	0.10	0.00	0.35	0.38
	HumanXORVal3	0.1	0.10	0.10	0.20	0.60	0.90	0.00	0.00	0.10	0.25
XOR Value	HumanXORVal4	0.5	0.40	0.40	0.40	0.70	0.92	0.10	0.00	0.40	0.43
	HumanXORMD5Hash0	1.0	1.0	1.0	0.90	1.0	0.93	0.90	0.95	0.98	0.96
	HumanXORMD5Hash1	1.0	1.0	1.0	1.00	1.0	0.96	1.00	1.00	1.00	0.99
	HumanXORMD5Hash2	1.0	1.0	1.0	1.00	1.0	0.96	1.00	1.00	1.00	0.99
	HumanXORMD5Hash3	1.0	1.0	1.0	1.00	1.0	0.96	0.90	1.00	1.00	0.98
XOR Value MD5	HumanXORMD5Hash4	1.0	1.0	1.0	1.00	1.0	0.95	1.00	0.95	1.00	0.99

Figure 4.10: NIST randomness test results for human player (pass value for NIST test = 0.8)



Data (AI Agent)	Data File	NIST Tests Applied									
		Frequency (Monobit) Test	Frequency Test within a Block	Cumulative Sums (Cusum) Test	Runs Test	Test for the Longest Run of Ones in a Block	Non-overlapping Template Matching Test	Approximate Entropy Test	Serial Test	Median	Average
	AITotVal0	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.12
	AITotVal1	0.00	0.00	0.00	0.00	0.00	0.95	0.00	0.00	0.00	0.12
	AITotVal2	0.00	0.00	0.00	0.00	0.00	0.97	0.00	0.00	0.00	0.12
	AITotVal3	0.00	0.00	0.00	0.00	0.00	0.96	0.00	0.00	0.00	0.12
Total Value	AITotVal4	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.12
	AITotMD5Hash0	0.80	1.00	0.80	1.00	0.60	0.90	0.40	0.35	0.80	0.73
	AITotMD5Hash1	1.00	1.00	1.00	0.90	0.90	0.91	0.60	0.60	0.90	0.86
	AITotMD5Hash2	0.90	1.00	0.90	1.00	0.90	0.90	0.60	0.50	0.90	0.84
	AITotMD5Hash3	0.60	1.00	0.60	0.70	0.40	0.89	0.30	0.35	0.60	0.61
Total Value MD5	AITotMD5Hash4	1.00	1.00	1.00	1.00	0.50	0.89	0.50	0.60	0.95	0.81
	AIXORVal0	0.50	0.50	0.60	0.60	0.80	0.83	0.50	0.30	0.55	0.58
	AIXORVal1	0.60	0.60	0.55	0.50	0.50	0.84	0.10	0.15	0.53	0.48
	AIXORVal2	0.70	0.60	0.70	0.50	0.70	0.94	0.50	0.25	0.65	0.61
	AIXORVal3	0.60	0.50	0.55	0.60	0.80	0.82	0.40	0.40	0.58	0.58
XOR Value	AIXORVal4	0.50	0.60	0.50	0.70	0.60	0.84	0.50	0.35	0.55	0.57
	AIXORMD5Hash0	1.00	1.00	1.00	1.00	1.00	0.96	1.00	1.00	1.00	0.99
	AIXORMD5Hash1	1.00	1.00	1.00	1.00	1.00	0.96	1.00	1.00	1.00	1.00
	AIXORMD5Hash2	1.00	1.00	1.00	0.90	1.00	0.96	1.00	1.00	1.00	0.98
	AIXORMD5Hash3	1.00	1.00	1.00	1.00	1.00	0.97	1.00	1.00	1.00	1.00
XOR Value MD5	AIXORMD5Hash4	1.00	0.90	1.00	1.00	1.00	0.96	1.00	1.00	1.00	0.98

Figure 4.11: NIST randomness test results for AI agent (pass value for NIST test = 0.8)

For executing the NIST tests, each individual bit stream needed to be divided into chunks of bits of equal length, also referred to as "sequences of bit streams" [60]. The NIST tests are applied on each of these sequences and the count of sequences passing the tests determines the randomness of the bit stream. NIST recommends that for “statistically meaningful results at least 55 sequences must be processed”. Since our bit streams had fewer bits, we decided to use the maximum number of sequences we could use. Also we considered the fact that the length of the sequence should be such that we can accommodate as many tests as possible from the NIST tests based on the input size recommendation for the tests. We fixed the number of sequences to be 10 for all our bit streams. That is, our bit streams will be divided into 10 chunks or “sequences” of bits of equal length. This was to ensure that we could apply the selected tests consistently across all bit streams. Obviously, in light of NIST’s recommendation above, our results should be treated as suggestive rather than conclusive.

We ran the selected 8 NIST tests on 40 bit streams (includes the bit streams collected for the human player and AI agent). As per NIST guidelines and also mentioned in their test reports, “The minimum pass rate for each statistical test with the exception of the random excursion (variant) test is approximately = 8 for a sample size = 10 binary sequences”. Consider the “Non-overlapping Template Matching Test” from NIST. For a parameter set in the test, relevant for the size of our binary sequence, the test uses 148 templates. Hence for this test, the total number of tests executed for 10 binary sequences amounts to 1480. We computed the ratio of the total number of tests passed out of 1480 tests and then compared it to the pass rate which is 0.8 (8/10 binary sequences) to determine pass or failure in this test. The results of the tests are given in Figure 4.10 for the human player and in Figure 4.11 for the AI agent.

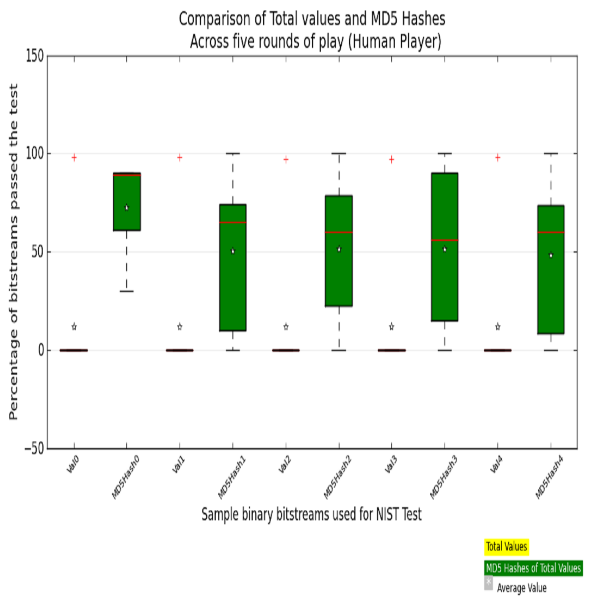
Based on the test results, we did our randomness analysis to determine how to extract randomness from Super Mario game. With random output from the game we can produce more diverse obfuscated code versions of the original code input.

We analyzed the randomness between the original data and the MD5 hash value of the data. The results showed that the MD5 values are more random than the original data. Referring to the boxplot graphs in Figure 4.12 (page 86), the MD5 hash of the original data passes more randomness tests than the original data. This is not a surprising result, but it helps put the results of non-MD5 randomness sources in context; it also highlights the choice of MD5 inputs, since not all MD5 results pass the NIST tests. From the graphs, it can also be concluded that the MD5 hash of XOR values for both human player and AI agent is more successful than total values in randomness tests.

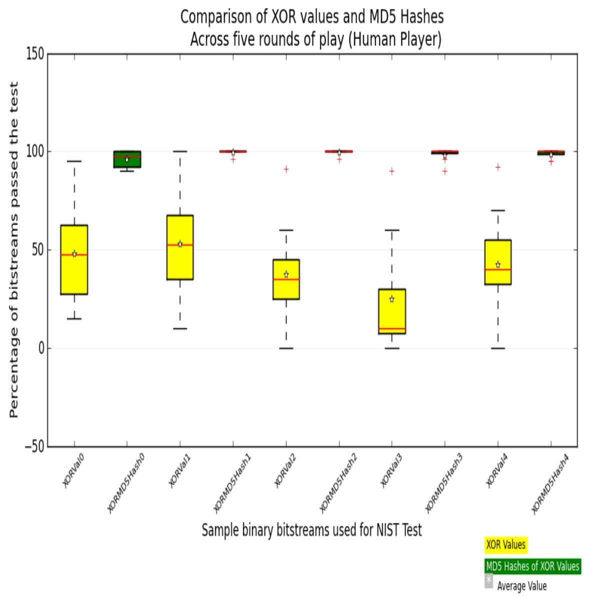
We compared the randomness of the values produced by XOR function versus the normal addition of the total count of game elements. It could be seen that XOR values (normal as well as MD5 hashes) pass more randomness tests than total values of game elements (Figure 4.13 on page 87). One possible interpretation of this result is that the relative distance of x and y coordinates of Mario are more random than the total count of game elements at any time in the game. So possibly, using the x and y coordinates of the player character will be a good source of randomness for obfuscating source code.

We examined the randomness generated by the human player and AI agent. The results are interpreted in Figure 4.14 (page 88). As can be seen from the graphs, there is no notable difference between the behaviors of human player and AI agent in terms of passing the randomness tests. Hence it may be difficult to prefer one above the other as being more random.

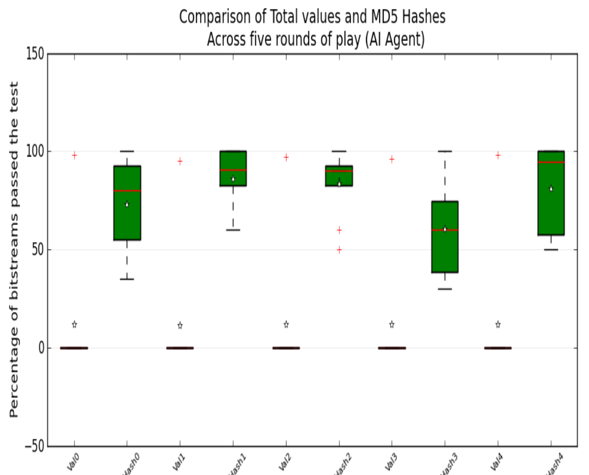
Finally, based on our tests with the NIST test suite and the input bit streams generated by playing Super Mario, we could predict that using player movement as compared to game elements and using MD5 hash values may provide more random data for code obfuscation.

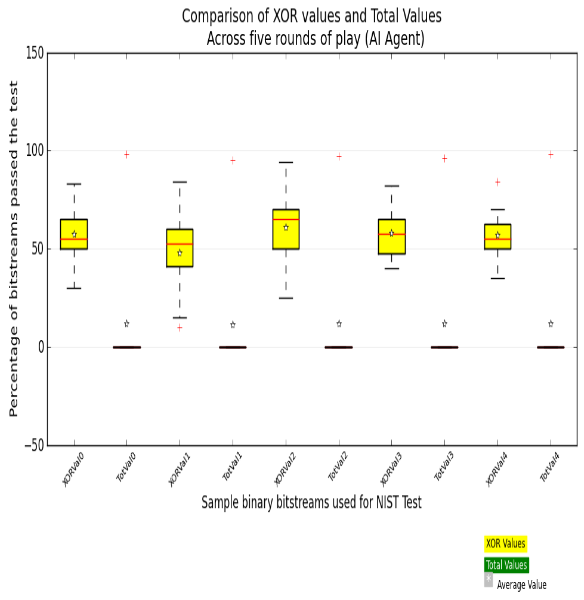


(a) Human player - total values vs. MD5 hash

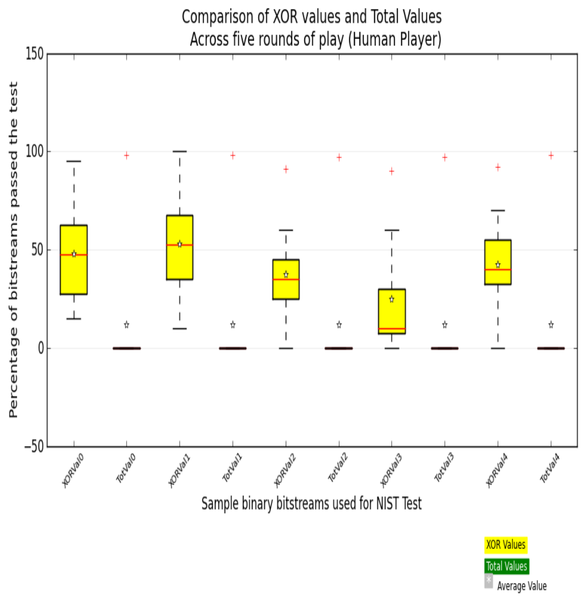


(b) Human player - XOR values vs. MD5 hash

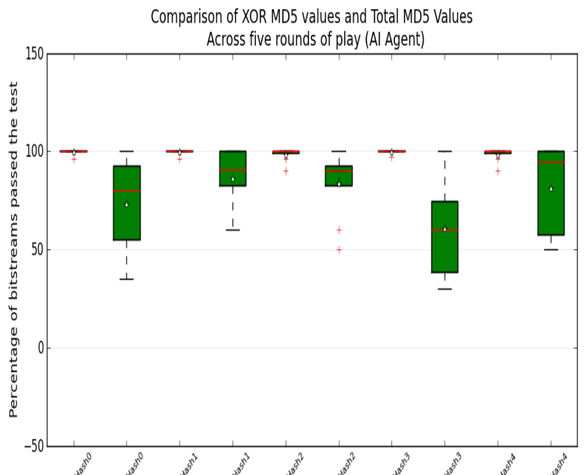


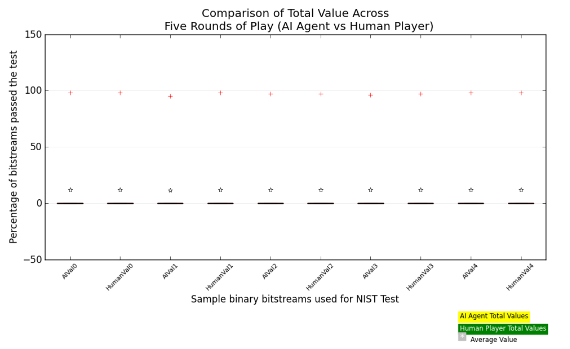


(a) AI agent - XOR values vs. total values

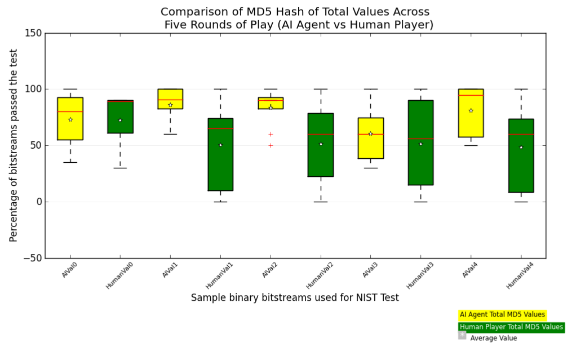


(b) Human player - XOR values vs. total values

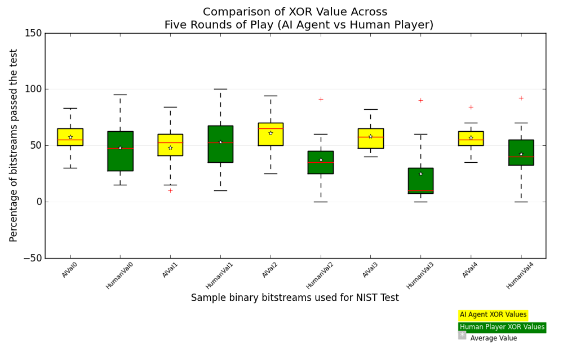




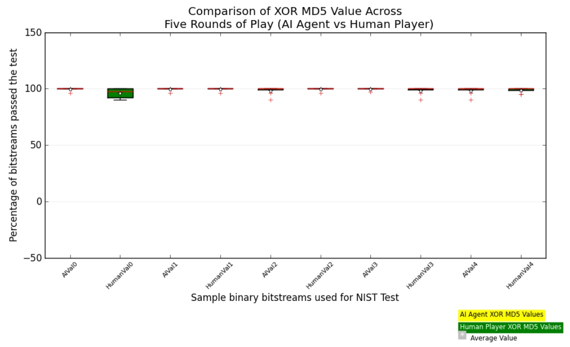
(a) Total value - AI agent vs. human player



(b) Total value MD5 hash - AI agent vs. human player



(c) XOR value - AI agent vs. human player



(d) XOR value MD5 hash - AI agent vs. human player

Figure 4.14: Random testing - human player vs. AI agent

Limitations. We faced certain limitations while extracting randomness from Super Mario for code obfuscation. One round of gameplay did not generate enough data for obfuscation, so we had to append data from several rounds of gameplay to produce the obfuscated version.

We were extracting individual values such as the relative distance of Mario in x and y coordinates to use in obfuscations such as spaghetti code generation. This type of obfuscation produced labeled GOTO blocks slicing the obfuscated code in chunks of code blocks. Another limitation we faced is to have sufficient distinct relative values to produce the required number of blocks to obfuscate the code. Sometimes, if distinct relative values of x and y coordinate are less, then the number of code blocks generated will be few. This will defeat the purpose of hiding the data flow significantly using spaghetti code.

It could also be seen from the analysis of randomness testing that using statistics based on game elements or player movement alone may not provide random data. We may need to compute some higher order functions or use MD5 to get a sufficiently random source of data.

4.4 Summary

After designing the previous game to obfuscate code, we decided to check if we can use the same approach with the existing games. We selected Super Mario (as used for the Mario AI 2009 competition) for our analysis as it provided similar flexibility for player action and difficulty level. Apart from that, we could use the same type of data we used in our previous game for obfuscation purposes. We also had access to the source code of the game to make necessary modifications. We were able to produce obfuscated code using Super Mario.

In Super Mario, we had less flexibility in terms of mapping game elements to code elements. So we had to shuffle the obfuscated code to achieve obfuscations such as code re-ordering.

We wanted to achieve diverse versions of obfuscated code. In the previous game, we were dynamically generating the game levels to induce randomness in player actions. This version of Super Mario implemented a procedural content generation engine based on a random seed value to generate the game levels [57]. We plotted player movement across different levels and the variation attained between the levels was not significant. So we looked at other sources to achieve the required randomness.

Game elements such as the coins collected by Mario, the enemies killed, and the number of lives available to Mario in the game could vary during the game play. Similarly, we considered the player movement and time taken to complete the level as another source of variation in the game. We applied MD5 hash functions to the values obtained from these sources to create another stream of random data. To get some idea of randomness of these sources we used the standard NIST test suite for testing random sequences [60].

Based on the test results of the NIST test suite, it appears that MD5 hash values are the most effective source of randomness. Using player movement may provide more randomness as compared to using statistics based on game elements.

Chapter 5

Conclusions

Our research was based on a novel idea for code obfuscation, to use humans playing video games. Our work has demonstrated that it is possible to make humans obfuscate code while playing video games. Recently, humans have been used to solve quite a few problems in science and technology using the concept of crowd sourcing. In such cases, a crowd sourced group consists of people with specific skills required to solve a problem. Our motivation was to abstract the need to have programming skills required for code obfuscation. So we opted for video games as the medium for crowd sourcing. The outcome of our research shows that humans playing video games could obfuscate a simple code to a version which is more difficult to understand.

5.1 How can video games be used for code obfuscation?

Video games can be customized to represent code. For example, the elements of the game could map to code elements. Player interaction with the game elements could be translated to code operations. Constraints in the game could be presented in a way to define the code flow. Hence video games provide a possible setup to generate code during gameplay. Also this will help us to achieve the purpose to trick the player into programming code while they believe they are playing a video game. Moreover video games on the Internet are accessible to a large virtual crowd and hence it may not be difficult to get players playing the video games.

5.2 Obfuscating code using video games

Our research consists of two proofs-of-concept to determine the feasibility of code obfuscation using humans playing video games. First, through a video game developed to generate obfuscated code. Second, using an existing video game to generate obfuscated code. We used obfuscation techniques involving computation transformations: dead code insertion, adding redundant operands, control flow obfuscations such as spaghetti code, and code re-ordering. Dead code insertion adds irrelevant code to the source code to abstract the control flow. Redundant operands are used to obfuscate arithmetic expressions. Spaghetti code hides the control flow by breaking the flow into blocks of code labeled with "GOTO" blocks. Code re-ordering shifts unrelated code snippets in random order. The source code we used for our analysis is straight line code performing arithmetic calculations. The rationale to use math operations was that math functions could be used to define complex operations as evident in the case of one instruction set computers (OISC).

We developed a simple video game known as "Popper" which was customized to produce obfuscated code during gameplay. Popper is a 2D platform game where the player character jumps between dual-colored platforms to pop all colored balloons. The colored gems on the game screen decide whether a balloon could be popped. If two gems match the colors of the dual-colored platform, then the balloon can be popped. The outcome of the balloon popping action (if it is not the last balloon to be popped), is another new gem with the same color as the popped balloon. Enemy characters such as roaming ghosts and jumping monkeys are used to add conflict in the gameplay. Neon tokens are used in the game to make the game objectives achievable. Collisions between the player and neon tokens generate a ladder to jump to the nearest platform and also to earn lives lost.

The source code is split into a sequence of binary operations (two operands and one operator). The variables in the code are represented by colored balloons. The player interaction with the colored balloons triggers the binary operations in the code. The constraints of the puzzle (color gems matching the colors of the platform) maintains the flow of the source code.

The obfuscating code transforms are spawned by incidents such as the player colliding with the neon tokens in the game or the dual-colored platforms. The relative motion values (x and y coordinates) of the player character are used individually as redundant operands to modify a variable and as labels for spaghetti code. The total value of the coordinates is used as a value assigned to a junk variable and added to the code generated by the game. The code re-ordering obfuscation is achieved by introducing multiple ways to play the game. To vary player movement, game levels were generated dynamically using a procedural content generation engine. This produced diverse versions of obfuscated code for the given source code.

The next step was to analyze if we can use an existing video game to obfuscate code. Our objective was to apply the obfuscation techniques used in “Popper” to an existing game to test the feasibility of generating obfuscated code. The Super Mario version from the Mario AI competition of 2009 was selected for the analysis because: the source code is available publicly; Super Mario is also a platform game like “Popper”, so could use the same approach for obfuscating code; and it also uses a procedural content generation engine based on a random seed to generate the game levels. The challenge was we could not map the game elements to code elements, unlike “Popper”. Neither could we introduce constraints in the game to maintain the control flow of the code. Hence we used the randomness produced in the game by the interaction of the player (Mario in this game) with game elements for the purpose of obfuscation.

Super Mario is a side-scrolling 2D platform game. The player character, Mario, moves through elevated platforms collecting coins. The conflicts in the game are added by moving enemies and the gaps in between the platforms. The game score is determined by the time taken to complete the level, the number of coins collected and enemies killed.

The three factors considered as a source of randomness in the game were the relative movement of the player (x and y coordinates), the statistics of the game elements (number of lives left, number of coins collected and enemies killed by Mario), and the current time in the game. The data corresponding to these factors were logged at regular intervals. To compare these random sources in the game, we used tests for randomness provided by NIST.

The logged data were processed into four sets of binary sequences for randomness testing: XOR of relative x, y and absolute time; the MD5 hash of the XOR value; the total of game statistics; the MD5 hash value computed from the total value. We wanted to figure out a number of ways to extract randomness from the original random data collected from the gameplay of Super Mario. Hence we evaluated if cryptographic functions applied on the logged data will be more random than the data itself. Our analysis of the test results from NIST showed that MD5 hash values are more random than the data used to compute the hashes, and XOR values were more random than the total value of game statistics. The main objective of Super Mario is to complete the level as fast as possible rather than collecting as many coins or killing as many enemies. This is probably the reason that relative movement of Mario is more random than the game statistics. We used the binary stream of the XOR values for obfuscation purposes.

Obfuscating transforms in case of Super Mario are obtained from the data logged as well as reading random bits from the binary sequence of XOR values. Like "Popper", we used the relative motion values (y coordinate) of Mario as labels for spaghetti code. The current time of the game is logged and used for dead code insertion. The junk variable to be used is determined by reading random bits from the binary sequence. The data collected at each interval is summed up and used as a redundant operand to modify variable values. We did not have the flexibility to induce multiple ways to complete the level in Super Mario. Hence code re-ordering was achieved by shuffling the code after generating the obfuscated output.

We compared the randomness produced by the winning AI agent of the Mario AI 2009 competition with respect to a human player. We did not find any significant difference between the two players in terms of random sequence generation.

5.3 Limitations

We had a few challenges while implementing the projects for our proofs of concept.

For the technical implementation of our research, we used a simple arithmetic expression computing method as a test case for code obfuscation. It remains to be seen if more complex code such as loop statements and classes with methods could be obfuscated using this research idea.

The types of obfuscations used for this work are computation transformations. It is probable that the obfuscations generated by this work could possibly be deobfuscated by deobfuscators. We ran some of our obfuscated code (Listing 3.5 on page 57) through the gcc compiler (version 4.8.3 20140911) running on a 64-bit Linux machine. We also generated the optimized version of the obfuscated code using the "-O3" flag of gcc which is the highest level of optimization provided by gcc. The number of lines of assembly code generated by gcc for the obfuscated code without the optimization flag is 161. The optimized version of the code generated with the optimization flag is reduced to 26 lines of code. We found that the compiler removed all obfuscating transformations from the obfuscated code. We compared and found that the optimized version of original code and the optimized version of the obfuscated code are identical (Listing 5.1). This suggests that we need to implement more complex code obfuscation procedures which are resilient to deobfuscators and code optimizers using gameplay from video games for code obfuscations.

```
.file   "ObfuscatedCode.c"
 .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
 .string "res = %d"
 .section        .text.startup,"ax",@progbits
 .p2align 4,,15
 .globl  main
 .type   main, @function
main:
.LFB33:
 .cfi_startproc
 subq    $8, %rsp
 .cfi_def_cfa_offset 16
 movl    $2, %esi
 movl    $.LC0, %edi
 xorl    %eax, %eax
 call    printf
 xorl    %eax, %eax
 addq    $8, %rsp
 .cfi_def_cfa_offset 8
 ret
 .cfi_endproc
.LFE33:
 .size   main, .-main
 .ident  "GCC: (GNU) 4.8.3 20140911 (Red Hat 4.8.3-7)"
 .section        .note.GNU-stack,"",@progbits
```

Listing 5.1: GCC optimized obfuscated code

We need to analyze if it is possible to incorporate other types of obfuscations such as using opaque predicts or extending loop conditions by customizing the gameplay of video games [1].

Crowdsourcing has been successfully used in certain cases to improve efficiency or design better solutions as compared to machines doing the same tasks. For example, in the game FoldIt [29], humans have been useful in finding solutions for protein folding which was computationally challenging due to the large search space of protein structures. We mentioned superoptimization [14] in our "Related Work" chapter. This, as observed by Massalin in her work [14], could make the program logic hard to understand. Hence this technique can also be used as a source to achieve obfuscating transformations. But there is not an efficient method available to use this technique due to the large search space of potential equivalent instructions. Hence, it may be possible to design a game to map these instructions to game elements. Then, using player inputs from the game, it could be possible to design a transformed version of the original code which is difficult to understand.

Back to our thesis, our first attempt was to generate obfuscated code by developing a customized platform video game. Hence for our second proof-of-concept, we restricted ourself to a video game belonging to the same genre, which is platform games. We need to check if it is possible to use games from different genres for code obfuscation purposes.

The obfuscated code generated was much longer in size as compared to the original source code. As observed in this paper [1], "A Taxonomy of Obfuscating Transformations", "increase in overall program size" is a necessary condition for a "potent obfuscating transformation". The paper defines that the quality of obfuscation depends on "the combination of potency, resilience and cost" of the transformation. If the obfuscated program is longer, it may conflict with the cost of the transformation. That is, longer code will take more time to execute than the shorter source code.

In the case of using game parameters for obfuscation, we used relative values of y-coordinates of the player character in both games for "GOTO" labels. We were faced with difficulties regarding this choice. Such as, often the relative values of player character movement logged from the game will be duplicated. Hence we had a smaller number of distinct values to be used for obfuscation.

5.4 Future Work

Since we are considering humans playing games as a source of randomness, we could analyze games of other genres to extract randomness instead of restricting our efforts to platform games. We can analyze what type or types of obfuscation can be more potent to confuse humans, be resilient to de-obfuscators, but be less costly in terms of memory usage and execution. As humans playing video games could generate a sequence of random numbers to be used for obfuscation, it may not be required to map obfuscating transforms to game elements. This may also help us achieve more complex obfuscations using this method.

We have analyzed the gameplay of both human players and an AI agent for Super Mario. We did not find any significant difference between the two types of players. This may indicate that AI agents could also be used like human players to generate randomness while playing video games. But for "Popper", the game we developed, it may be worthwhile to develop an AI agent and do the same analysis. Since in the case of "Popper", the procedurally generated game levels are more dynamic than the game levels in Super Mario.

We tested the random sequence generated by humans playing video games with the NIST test suite. But we did not have enough data sets to conduct all the random tests of the test suite. So in future, another milestone could be to generate enough data set to use all tests for randomness testing. Also, we could compare the strength of randomness versus another established pseudo-random number generator.

We were able to achieve our goal of obfuscating code through video games, one game which has been strategically developed, and an existing video game. It will be beneficial to do a comparison of the quality of obfuscating transforms between the two approaches.

Another possible area of future work will be to design a defensive technique against this approach to prevent adversaries for using this idea for malicious purposes.

Bibliography

[1] C.S. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Technical Report 148, Dept. of Computer Science, Univ. of Auckland, July 1997

[2] C.S. Collberg and C. Thomborson. "Watermarking, tamper-proofing and obfuscation-tools for software protection." Software Engineering, IEEE Transactions on, 28:735-746,2002.

[3] J. Aycock, J.M.G. Cardenas and D.M.N. de Castro. "Code obfuscation using pseudo-random number generators." In Computational Science and Engineering, 2009, CSE '09. International Conference on Computational Science and Engineering volume 3, pages 418-423, Aug 2009.

[4] P. Porras, H. Saidi, and V. Yegneswaran, "An analysis of Conficker's logic and rendezvous points," SRI International Technical Report, 2009, last update 19 March 2009. [Online]. Available at: `http://mtc.sri.com/Conficker`

[5] J. Riordan and B. Schneier, "Environmental key generation towards clueless agents," In Mobile Agents and Security (LNCS 1419), 1998, pp. 1524

[6] Ilsun You and Kangbin Yim, "Malware Obfuscation Techniques: A Brief Survey", 2010 International Conference on Broadband, Wireless Computing, Communication and Applications

[7] M. Schiffman, "A Brief History of Malware Obfuscation", Available at `http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2/`, Feb. 2010.

[8] W. Wong and M. Stamp, “Hunting for Metamorphic Engines” Journal in Computer Virology, vol. 2, no. 3, pp. 211-229, Dec. 2006

[9] M. Christodorescu and S. Jha, “Static Analysis of Executables to Detect Malicious Patterns,” Proceedings of the 12th conference on USENIX Security Symposium, Vol. 1, pp. 169-186, Aug. 2003.

[10] E. Konstantinou, “Metamorphic Virus: Analysis and Detection,” RHUL-MA-2008-02, Technical Report of University of London, Jan. 2008. Available at `http://www.rhul.ac.uk/mathematics/techreports`

[11] Cooper, Keith D. and Torczon, Linda, Engineering a Compiler, Morgan Kaufmann, 2004, ISBN 1-55860-699-8

[12] Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane and Arun Lakhotia, “The Design Space of Metamorphic Malware” Proceedings of the 2nd International Conference on Information Warfare and Security (ICIW 2007). pp. 241-248, (2007).

[13] Ramneek, Puri (2003-08-08). “Bots and Botnet: An Overview” (PDF). SANS Institute. Retrieved 2 July 2014

[14] Massalin, H.: Superoptimizer: A look at the smallest program. In: ASPLOS-II: Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems, pp. 122-126. IEEE Computer Society Press, Los Alamitos (1987)

[15] J.M. Borello and L. Me, “Code obfuscation techniques for metamorphic viruses,” Journal in Computer Virology, vol. 4 (no.3), 2008, pp. 211-220

[16] D. Bruschi, L. Martignoni, and M. Monga, “Code normalization for self-mutating malware,” IEEE Security and Privacy, vol. 5,2007, pp. 46-54.

[17] Howe, Jeff (June 2, 2006). "Crowdsourcing: A Definition". Crowdsourcing Blog. Retrieved January 2, 2013.

[18] Brabham, Daren (2008), "Crowdsourcing as a Model for Problem Solving: An Introduction and Cases" (PDF), Convergence: The International Journal of Research into New Media Technologies 14 (1): 7590, doi>10.1177/1354856507084420

[19] Lee, J. H., Karlova, N., Clarke, R. I., Thornton, K., & Perti, A. (2014). "Facet Analysis of Video Game Genres.", In iConference 2014 Proceedings (p. 125 - 139). doi>10.9776/14057

[20] F. Perriot, P. Ferrie (2004). "Principles and Practise of X-Raying", Virus Bulletin Conference. [Online]. Available at: `http://pferrie.tripod.com/papers/x-raying.pdf`

[21] M. Jacob, M. Jakubowski, P. Naldurg, C. Saw, and R. Venkatesan." The superdiversifier: Peephole individualization for software protection." In K. Matsuura and E. Fujisaki, editors, Advances in Information and Computer Security, volume 5312 of Lecture Notes in Computer Science, pages 100120. Springer Berlin / Heidelberg, 2008.

[22] Safire, William (February 5, 2009). "On Language - Fat Tail". New York Times Magazine. Retrieved July 14, 2014.

[23] Microwork and Microfinance, Leila Chirayath Janah, August 25, 2009 Social Edge. Retrieved 2013-01-03: `http://www.socialedge.org/blogs/samasourcing/archive/2009/08/25/microwork-and-microfinance`

[24] P.C. (2011) "Jobs of the future", The digital economy, Retrieved 2013-01-03: `http://www.economist.com/blogs/schumpeter/2011/04/digital_economy`

[25] Amazon Mechanical Turk Beta, Requester Retrieved 2014-14-07: `https://requester.mturk.com`

[26] Jason Pontin, The New York Times (2007), "Artificial Intelligence, With Help From the Humans", Retrieved 2014-14-07: `http://www.nytimes.com/2007/03/25/business/yourmoney/25Stream.html`

[27] Luis von Ahn, Ben Maurer, Colin McMillen, David Abraham and Manuel Blum (2008). "reCAPTCHA: Human-Based Character Recognition via Web Security Measures" (PDF). Science 321 (5895): 14651468. doi:10.1126/science.1160379.PMID 18703711

[28] Frederic Lardinois (2012) TechCrunch, "reCaptcha Founders Language Learning Site Duolingo To Open To The Public On June 19", Retrieved 2014-14-07: `http://techcrunch.com/2012/05/22/recaptcha-founders-language-learning-site-duolingo-to-open-to-the-public-on-june-19/`

[29] Seth Cooper, Firas Khatib, Adrien Treuille, Janos Barbero, Jeehyung Lee, Michael Beenen, Andrew Leaver-Fay, David Baker, Zoran Popovic and Foldit players, (2010), "Predicting protein structures with a multiplayer online game", Vol 466, Nature. Retrieved 2014-14-07: `http://www.nature.com/nature/journal/v466/n7307/pdf/nature09304.pdf`

[30] Brabham, Daren (2008), "Crowdsourcing as a Model for Problem Solving: An Introduction and Cases" Convergence: The International Journal of Research into New Media Technologies 14 (1): 7590 : `http://www.clickadvisor.com/downloads/Brabham_Crowdsourcing_Problem_Solving.pdf`

[31] Martin Hyland and Andrea Schalk. 2002. "Games on Graphs and Sequentially Realizable Functionals." In Proceedings of the 17th Annual IEEE Symposium

on Logic in Computer Science (LICS '02). IEEE Computer Society, Washington, DC, USA, 257-264.

[32] Donald E. Knuth (1997) The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd edition). Addison-Wesley Professional

[33] Dan Biebighauser: Testing Random Number Generators. University of Minnesota, 2000.

[34] J. Levine. Re: [Asrg] A CAPTCHA that automatically detects and neutralizes attacks. `http://www.ietf.org/mail-archive/web/asrg/current/msg12051.html`, 13 May 2005. Asrg mailing list.

[35] Ahn, Luis von; Blum, Manuel; Hopper, Nicholas J.; Langford, John (2003). "CAPTCHA: Using Hard AI Problems for Security" Advances in Cryptology EUROCRYPT 2003. Lecture Notes in Computer Science 2656

[36] "Data protection: "Junk" e-mail costs internet users 10 billion a year worldwide - Commission study" Europa.eu. Retrieved 2014-07-17.

[37] Alexander Zook and Mark O. Riedl. Game Conceptualization and Development Processes in the Global Game Jam. Proceedings of the FDG 2013 Workshop on the Global Game Jam, Chania, Crete, Greece, 2013.

[38] Elina M. I. Ollila , Riku Suomela , Jussi Holopainen, Using prototypes in early pervasive game development, Computers in Entertainment (CIE), v.6 n.2, April/June 2008 [doi>10.1145/1371216.1371220]

[39] R. Halprin and M. Naor, "Games for extracting randomness," in Proceedings of SOUPS '09, 2009.

[40] Ferrucci, F., Tortora, G. and Vitiello, G. 2002. Visual Programming. Encyclopedia of Software Engineering.

[41] Conway, M., Audia, S., Burnette, T., Cosgrove, D. and Christiansen, K. (2000) Alice: lessons learned from building a 3D system for novices. Proceedings of CHI 2000, pp. 486-493.

[42] A. Fowler, T. Fristce, M. MacLauren, “Kodu Game Lab: a programming environment” in The Computer Games Journal Ltd 2012-2014

[43] Ken Kahn, “A Computer Game to Teach Programming” in the Proceedings of the National Educational Computing Conference 1999

[44] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. 2013. Procedural content generation for games: A survey. ACM Trans. Multimedia Comput. Commun. Appl. 9, 1, Article 1 (February 2013), 22 pages. DOI=10.11452422956.2422957

[45] Jones, Joel. Abstract Syntax Tree Implementation Idioms. (overview of AST implementation in various language families), Proceedings of the 10th Conference on Pattern Languages of Programs PLoP2003, (2003) Retrieved on 2014-01-08: `http://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf`

[46] Amanda Greenslade, ”Gamespeak: A glossary of gaming terms”. Specusphere. Archived from the original on 2007-02-19. Retrieved 2014-01-08 : `http://en.wikipedia.org/wiki/Platform_game`

[47] Mavaddat, F.; Parhami, B. (October 1988). “URISC: The Ultimate Reduced Instruction Set Computer”. Int’l J. Electrical Engineering Education (Manchester University Press) 25 (4): 327334. Retrieved 2014-08-06

[48] Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, Ricardo Motta (November 5, 1996). Retrieved 2014-08-07: `http://www.w3.org/Graphics/Color/sRGB`

[49] (Technical report). Adobe Systems Incorporated. 13 May 2005. Retrieved 2014-08-07: `http://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf`

[50] James C. King "Why Color Management", Adobe Systems Incorporated. Retrieved 2014-08-07: `http://www.color.org/whycolormanagement.pdf`

[51] Technical Guide. Adobe. Retrieved 2014-08-07: `http://dba.med.sc.edu/price/irf/Adobe_tg/models/cielab.html`

[52] Technical Guide. Adobe. Retrieved 2014-08-07: `http://dba.med.sc.edu/price/irf/Adobe_tg/models/ciexyz.html`

[53] Hunter, Richard Sewall (December 1948). "Accuracy, Precision, and Stability of New Photo-electric Color-Difference Meter". JOSA 38 (12): 1094. (Proceedings of the Thirty-Third Annual Meeting of the Optical Society of America)

[54] Hoffman, G. (2010). CIE colour space. Retrieved 2014-08-07: `http://www.fho-emden.de/~hoffmann/ciexyz29082000.pdf`

[55] Lindbloom, Bruce Justin. "Delta E (CIE 1994)" Brucelindbloom.com. Retrieved 2014-08-07: `http://www.brucelindbloom.com/index.html?Eqn_DeltaE_CIE94.html`

[56] Debray, S. K., Evans, W., Muth, R., and De Sutter, B. 2000. Compiler techniques for code compaction. ACM Trans. Program. Lang. Syst. 22, 2 (Mar. 2000), 378-415

[57] Togelius, J.; Karakovskiy, S.; Baumgarten, R., "The 2009 Mario AI Competition," Evolutionary Computation (CEC), 2010 IEEE Congress on , vol., no., pp.1,8, 18-23 July 2010

[58] Yong-Xia, Zhao; Zhao Yong-Xia; Ge, Zhen; Zhen Ge, "MD5 Research" 2010 Second International Conference on Multimedia and Information Technology, 2010, ISBN 0769540082, Volume 2, pp. 271 - 273

[59] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13, 4 (October 1991), 451-490

[60] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, S. Vo, A statistical test suite for random and pseudorandom number generators for cryptographic applications, National Institute of Standards and Technology (NIST), special publication 800-22, August 2008.

Appendix A

Procedural Content Generation Algorithm

Algorithm 6 Procedural Content Generation Engine

```
 1: procedure PCGENGINE($room\_height$,$room\_length$,$gems$,$tiles$)
 2:     The dimension of each block in the room is assumed to be 32x32
 3:     Divide $room\_height$ and $room\_length$ by 32
 4:     Use $room\_height$/32 and $room\_length$/32 as columns $grid\_cols$
        and rows $grid\_rows$ of a 2-D grid
 5:     The wall blocks in the game are of dimension 64x64
 6:     Update the 2-D grid structure with a generic symbol such as “x” in
        cols as $grid\_cols$-1 and rows as $grid\_rows$-1 due to the dimension
        of wall blocks
 7:     Update the border “x” of the grid with “W” which signifies walls
 8:     Now the game level structure will be updated in the 2-D grid with
        dimensions as number of grid rows $grid\_rows$-2 and grid columns
        $grid\_cols$-2 as “W” will occupy two rows and two columns
 9:     Define an arbitrary block height $blk\_height$ as 2 and block width as $blk\_width$
    as 6
10:     Define a block width or maximum x-coordinate of a block as
        $max\_x\_block$ and a minimum x-coordinate starting position of a
        block as $min\_x\_block$
11:     Initially $min\_x\_block$ is defined as 2 since the position of the block
        should start from 2 blocks from the start of the grid to account
        for wall blocks
12:     Block width $max\_x\_block$ set as 6 arbitrarily
13:     Define block height or maximum y-coordinate of a block
        as $max\_y\_block$ and a minimum y-coordinate of a block as
        $min\_y\_block$.
14:     Initially $max\_y\_block$ is set as $len(grid)$ - 1 which is the maximum
        row number in the grid, i.e. $grid\_rows$-2. This remains fixed
        throughout the algorithm
15:     Initially $min\_y\_block$ is set as $max\_y\_block$-$blk\_height$
16:     Define a maximum block width as $max\_blk\_width$ as $grid\_cols$-1
17:     Define a maximum block height as $max\_blk\_height$ as $min\_y\_block$
18:     while $max\_x\_block$ is less than $max\_blk\_width$ do
19:         for each y-coordinate $y$ in range of $min\_y\_block$ to $max\_y\_block$ do
20:             for each x-coordinate $x$ in range of $min\_x\_block$ to $max\_x\_block$ do
21:                 Update 2-D grid in position $x$, $y$ with “B”
22:             end for
```

```
23:         end for
24:         Set $blk\_height$ randomly from between range 1 to 3
25:         Update $min\_y\_block$ as $max\_y\_block$-$blk\_height$
26:         $max\_y\_block$ remains constant
27:         Update $min\_x\_block$ as $max\_x\_block$+1
28:         Set $max\_x\_block$ as $min\_x\_block$+a random value from between
            range 1 to 4
29:         if $min\_y\_block$ is less than $max\_blk\_height$ then
30:             Update $max\_blk\_height$ as $min\_y\_block$. This will give the mini-
                mum row number from where the blocks start
31:         end if
32:     end while
        // Finding the random x-coordinate position for colored gems in the game
33:     Update $gemx$ as any random value in range 2 to $grid\_cols$-4
34:     Update a variable $maxheightofgems$ as number of gems from $gems$
35:     Update $gemy$ as any random value in range 2 to $grid\_rows$-
        $maxheightofgems$-$max\_blk\_height$
36:     Update $x$ as $gemx$ and $y$ as $gemy$
        // Placing the gems in the grid
37:     for Each $gem$ in $gems$ do
38:         Update grid in $x$ and $y$ as col and row respectively with $gem$
39:         Update $x$ as $x$+1
40:         if $x$ is greater than $gemx$+1 then
41:             Update $y$ as $gemy$+1 and $x$ as $gemx$
42:         end if
43:     end for
        // Placing the tiles in the grid
44:     for each $tile$ in $tiles$ do
45:         Update $randx$ as random value in range 2 to $grid\_cols$-4
46:         Update $randy$ as random value in range 2 to $grid\_rows$-$max\_blk\_height$.
47:         while continue till the break condition in the following code
            is reached do
48:             if all positions surrounding the coordinate position $randx$ and $randy$
                has value “x” then
49:                 Break the loop
50:             end if
51:             Update $randx$ as random value in range 2 to $grid\_cols$-4
52:             Update $randy$ as random value in range 2 to $grid\_rows$-
                $max\_blk\_height$
53:         end while
54:     end for
55: end procedure
```

Appendix B

Obfuscated Code Output from Super Mario

Listing B.1: Obfuscated code with Super Mario

```
#include <stdio.h>
#include <memory.h
   >
#include <stdlib.h
   >
#include <string.h
   >

int main(int argc,
    char *argv[])
{
 int aa,bb,cc,dd;
 int b =  1;
 int c =  1;
 int num0 = 2;
 int x =  1;
 int y =  1;
 int res = 0;
 int res1 = 0;
 int b0 = 0;
 int res00 = 0;
 int num00 = 0;
 int res_01 = 0;
 int res10 = 0;
 int res_02 = 0;
 int res_03 = 0;
 int res_04 = 0;
 int res_05 = 0;
 int res_11 = 0;
 int b_1 = 0;
 int y0 = 0;
 int c0 = 0;
 int num_1 = 0;
 int c_1 = 0;
 int res_12 = 0;
 int res_06 = 0;
 int num_2 = 0;
 int res_07 = 0;
 int res_13 = 0;
 int res_14 = 0;
 int num_3 = 0;
 int res_08 = 0;
 int num_4 = 0;
 int res_09 = 0;
 int x0 = 0;
 int num_5 = 0;
 int res_15 = 0;
 int res_010 = 0;
 int res_011 = 0;
 int res_012 = 0;
 int res_013 = 0;
 int x_1 = 0;
 int x_2 = 0;
 int x_3 = 0;
 int x_4 = 0;
 int y_1 = 0;
 int res_014 = 0;
 int c_2 = 0;
 int num_6 = 0;
 int res_015 = 0;
 int res_016 = 0;
 int x_5 = 0;
 int y_2 = 0;
 int res_017 = 0;
 int b_2 = 0;
 int res_16 = 0;
 int c_3 = 0;
 int res_17 = 0;
 int res_018 = 0;
 int b_3 = 0;
 int num_7 = 0;
 int x_6 = 0;
 int res_18 = 0;
 int b_4 = 0;
 int c_4 = 0;
 int res_019 = 0;
 int num_8 = 0;
 int res_020 = 0;
 int y_3 = 0;
 int res_021 = 0;
 int b_5 = 0;
 int x_7 = 0;
 int res_022 = 0;
 int res_023 = 0;
 int res_024 = 0;
 int y_4 = 0;
 int res_025 = 0;
 int b_6 = 0;
 int x_8 = 0;
 int res_026 = 0;
 int res_027 = 0;
 int res_028 = 0;
 int y_5 = 0;
 int res_029 = 0;
 int b_7 = 0;
 int x_9 = 0;
 int res_030 = 0;
 int res_031 = 0;
 int num_9 = 0;
 int res_032 = 0;
 int res_033 = 0;
 int res_034 = 0;
 int c_5 = 0;
 int res_19 = 0;
 int num_10 = 0;
int res_20 = 0;
int res_035 = 0;
int b_8 = 0;
int res_21 = 0;
int res_22 = 0;
int res_23 = 0;
int b_9 = 0;
int y_6 = 0;
int res_036 = 0;
int b_10 = 0;
int res_24 = 0;
int y_7 = 0;
int res_037 = 0;
int b_11 = 0;
int y_8 = 0;
int res_038 = 0;
int res_039 = 0;
int c_6 = 0;
int b_12 = 0;
int x_10 = 0;
int c_7 = 0;
int y_9 = 0;
int b_13 = 0;
int num_11 = 0;
int res_040 = 0;
int res_041 = 0;
int b_14 = 0;
int b_15 = 0;
int res_042 = 0;
int y_10 = 0;
int b_16 = 0;
int res_043 = 0;
int res_044 = 0;
int x_11 = 0;
int res_25 = 0;
int res_26 = 0;
int b_17 = 0;
int res_27 = 0;
int res_045 = 0;
int y_11 = 0;
int b_18 = 0;
int x_12 = 0;
int x_13 = 0;
int res_28 = 0;
int res_046 = 0;
int res_047 = 0;
int res_048 = 0;
int b_19 = 0;
int c_8 = 0;
int res_049 = 0;
int num_12 = 0;
int res_29 = 0;
int num_13 = 0;
int res_30 = 0;
int res_31 = 0;
int b_20 = 0;
int y_12 = 0;
int res_050 = 0;
int res_32 = 0;
int res_051 = 0;
int res_052 = 0;
int y_13 = 0;
int b_21 = 0;
int b_22 = 0;
int x_14 = 0;
int c_9 = 0;
int res_053 = 0;
int y_14 = 0;
int x_15 = 0;
int x_16 = 0;
int b_23 = 0;
int res_054 = 0;
int res_055 = 0;
int res_33 = 0;
int res_056 = 0;
int y_15 = 0;
int y_16 = 0;
int res_057 = 0;
int b_24 = 0;
int x_17 = 0;
int res_058 = 0;
int res_059 = 0;
int num_14 = 0;
int b_25 = 0;
int num_15 = 0;
int res_060 = 0;
int res_061 = 0;
int b_26 = 0;
int res_34 = 0;
int res_062 = 0;
int res_063 = 0;
int y_17 = 0;
int res_064 = 0;
int c_10 = 0;
int b_27 = 0;
int num_16 = 0;
int res_065 = 0;
int res_066 = 0;
int res_35 = 0;
int x_18 = 0;
int y_18 = 0;
int res_067 = 0;
int b_28 = 0;
int y_19 = 0;
int res_068 = 0;
int res_069 = 0;
int c_11 = 0;
int y_20 = 0;
int res_070 = 0;
int b_29 = 0;
int c_12 = 0;
int res_071 = 0;
int num_17 = 0;
int res_36 = 0;
int x_19 = 0;
int res_37 = 0;
int res_38 = 0;
int b_30 = 0;
int y_21 = 0;
int res_072 = 0;
int res_39 = 0;
int b_31 = 0;
int c_13 = 0;
int num_18 = 0;
int x_20 = 0;
int y_22 = 0;
int res_073 = 0;
int res_40 = 0;
L13300003:
 cc = 2.0;
 b0 = b+1051.0;
 aa = 3.0;
 res00 = res
    +1039.0;
 dd = 4.0;
 num00 = num0
    +1040.0;
 cc = 5.0;
goto L00;
L00:
 bb = 3.0;
 res_01 = res00
    -1045.0;
 bb = 6.0;
 res10 = res1
    -1047.0;
 cc = 7.0;
 res_02 = res_01
    -1052.0;
 cc = 8.0;
 res_03 = res_02
    -1061.0;
goto L10;
L10:
 cc = 9.0;
 res_04 = res_03
    -1059.0;
 bb = 10.0;
 res_05 = res_04
    -1063.0;
 cc = 11.0;
 res_11 = res10
    -1064.0;
 bb = 12.0;
 b_1 = b0-1065.0;
goto L31243591;
L20:
 cc = 21.0;
 res_13 = res_12
    -1080.0;
 dd = 22.0;
 res_14 = res_13
    -1081.0;
 aa = 23.0;
 num_3 = num_2
    +1088.0;
 bb = 24.0;
 res_08 = res_07
    -1083.0;
goto L8304993;
```

```
L30:
 bb = 29.0;
 res_15 = res_14
    -1077.0;
 bb = 30.0;
 res_010 = res_09
    -1078.0;
 dd = 31.0;
 res_011 =
    res_010
    -1079.0;
 cc = 32.0;
 res_012 =
    res_011
    +1079.0;
goto L40;
L40:
 bb = 33.0;
 res_013 =
    res_012
    +1073.0;
 bb = 34.0;
 x_1 = x0-1077.0;
 dd = 35.0;
 x_2 = x_1
    +1081.0;
 dd = 36.0;
 x_3 = x_2
    -1082.0;
goto L50;
L50:
 dd = 37.0;
 x_4 = x_3
    -1083.0;
 dd = 38.0;
 y_1 = y0-1087.0;
 aa = 39.0;
 res_014 =
    res_013
    -1085.0;
 dd = 40.0;
 c_2 = c_1
    +1086.0;
goto L60;
L60:
```

```
 cc = 41.0;
 num_6 = num_5
    +1087.0;
 bb = 42.0;
 res_015 =
    res_014
    -1088.0;
 aa = 43.0;
 res_016 =
    res_015
    +1093.0;
 bb = 44.0;
 x_5 = x_4
    -1091.0;
goto L405925;
L70:
 dd = 49.0;
 c_3 = c_2
    +1094.0;
 bb = 50.0;
 res_17 = res_16
    +1095.0;
 aa = 51.0;
 res_018 =
    res_017
    -1096.0;
 cc = 52.0;
 b_3 = b_2
    +1097.0;
goto L5699997;
L80:
 y_3 = y_2
    -1106.0;
 res_022 =
    res_021
    -1110.0;
 aa = 62.0;
 res_021 =
    res_020
    +1107.0;
goto L90;
L90:
 cc = 63.0;
 b_5 = b_4
    +1108.0;
 aa = 64.0;
 x_7 = x_6
    -1109.0;
goto L100;
L100:
 cc = 65.0;
 bb = 66.0;
 res_023 =
    res_022
    +1111.0;
 dd = 67.0;
 res_024 =
    res_023
    +1112.0;
 dd = 68.0;
 y_4 = y_3
    -1113.0;
 aa = 69.0;
goto L110;
L110:
 res_025 =
    res_024
    +1114.0;
 cc = 70.0;
 b_6 = b_5
    +1115.0;
 aa = 71.0;
 x_8 = x_7
    -1116.0;
 cc = 72.0;
 res_026 =
    res_025
    -1117.0;
 bb = 73.0;
goto L120;
L120:
 res_027 =
    res_026
    +1118.0;
 dd = 74.0;
 res_028 =
    res_027
    +1119.0;
 dd = 75.0;
 y_5 = y_4
    -1120.0;
 cc = 76.0;
 res_029 =
    res_028
    +1121.0;
 dd = 77.0;
goto L62541504;
L130:
 num_9 = num_8
    -1126.0;
 dd = 82.0;
 res_032 =
    res_031
    -1127.0;
 dd = 83.0;
 res_033 =
    res_032
    +1128.0;
 cc = 84.0;
 res_034 =
    res_033
    -1129.0;
 aa = 85.0;
goto L140;
L140:
 c_5 = c_4
    +1130.0;
 cc = 86.0;
 res_19 = res_18
    +1146.0;
 cc = 87.0;
 num_10 = num_9
    +1140.0;
 cc = 88.0;
 res_20 = res_19
    -1134.0;
 bb = 89.0;
goto L150;
L150:
 res_035 =
    res_034
    +1142.0;
 aa = 90.0;
 b_8 = b_7
    +1148.0;
 aa = 91.0;
 res_21 = res_20
    -1141.0;
 dd = 92.0;
 res_22 = res_21
    -1148.0;
 cc = 93.0;
goto L160;
L160:
 res_23 = res_22
    +1143.0;
 aa = 94.0;
 b_9 = b_8
    +1144.0;
 bb = 95.0;
 y_6 = y_5
    +1149.0;
 dd = 96.0;
 res_036 =
    res_035
    +1140.0;
 dd = 97.0;
goto L170;
L170:
 b_10 = b_9
    -1153.0;
 cc = 98.0;
 res_24 = res_23
    -1154.0;
 aa = 99.0;
 y_7 = y_6
    -1146.0;
 aa = 100.0;
 res_037 =
    res_036
    +1155.0;
 aa = 101.0;
goto L180;
L180:
 b_11 = b_10
    +1158.0;
 aa = 102.0;
 y_8 = y_7
    -1148.0;
 dd = 103.0;
 res_038 =
    res_037
    -1158.0;
 bb = 104.0;
 res_039 =
    res_038
    +1159.0;
 bb = 105.0;
goto L190;
L190:
 c_6 = c_5
    -1161.0;
 aa = 106.0;
 b_12 = b_11
    -1162.0;
 aa = 107.0;
 x_10 = x_9
    +1163.0;
 cc = 108.0;
 c_7 = c_6
    -1164.0;
 dd = 109.0;
goto L200;
L200:
 y_9 = y_8
    +1175.0;
 bb = 110.0;
goto L210;
L210:
 b_13 = b_12
    -1166.0;
 aa = 111.0;
 num_11 = num_10
    -1167.0;
 aa = 112.0;
 res_040 =
    res_039
    +1164.0;
 bb = 113.0;
 res_041 =
    res_040
    -1176.0;
 dd = 114.0;
goto L220;
L220:
 b_14 = b_13
    +1170.0;
goto L230;
L230:
 aa = 115.0;
 b_15 = b_14
    -1166.0;
 aa = 116.0;
 res_042 =
    res_041
    +1167.0;
 dd = 117.0;
 y_10 = y_9
    -1168.0;
 cc = 118.0;
 b_16 = b_15
    +1169.0;
goto L240;
L240:
 dd = 119.0;
 res_043 =
    res_042
    +1161.0;
 dd = 120.0;
 res_044 =
    res_043
    +1173.0;
 bb = 121.0;
 x_11 = x_10
    -1168.0;
 aa = 122.0;
 res_25 = res_24
    -1169.0;
goto L250;
L250:
 dd = 123.0;
 res_26 = res_25
    +1173.0;
 cc = 124.0;
 b_17 = b_16
    -1191.0;
 aa = 125.0;
 res_27 = res_26
    +1179.0;
 cc = 126.0;
 res_045 =
    res_044
    -1184.0;
goto L260;
L260:
 bb = 127.0;
 y_11 = y_10
    +1181.0;
 aa = 128.0;
 b_18 = b_17
    +1181.0;
 aa = 129.0;
 x_12 = x_11
    +1191.0;
 dd = 130.0;
 x_13 = x_12
    -1183.0;
goto L35361786;
L270:
 dd = 135.0;
 b_19 = b_18
    +1189.0;
 aa = 136.0;
 c_8 = c_7
    +1202.0;
 bb = 137.0;
 res_049 =
    res_048
    +1208.0;
 dd = 138.0;
 num_12 = num_11
    +1201.0;
goto L280;
L280:
 dd = 139.0;
 res_29 = res_28
    +1202.0;
 aa = 140.0;
 num_13 = num_12
    +1203.0;
 bb = 141.0;
 res_30 = res_29
    -1204.0;
 cc = 142.0;
 res_31 = res_30
```

```
    +1200.0;
goto L290;
L290:
 aa = 143.0;
 b_20 = b_19
    +1201.0;
 aa = 144.0;
 y_12 = y_11
    +1202.0;
 bb = 145.0;
 res_050 =
    res_049
    -1203.0;
 aa = 146.0;
 res_32 = res_31
    -1204.0;
goto L300;
L300:
 cc = 147.0;
 res_051 =
    res_050
    -1213.0;
 cc = 148.0;
 res_052 =
    res_051
    +1214.0;
 dd = 2.0;
 y_13 = y_12
    -1045.0;
 b_21 = b_20
    -1038.0;
 aa = 4.0;
goto L310;
L310:
 b_22 = b_21
    +1039.0;
 cc = 5.0;
 x_14 = x_13
    -1044.0;
 dd = 6.0;
 c_9 = c_8
    -1055.0;
 bb = 7.0;
 res_053 =
    res_052
    -1050.0;
 cc = 8.0;
goto L320;
L320:
 y_14 = y_13
    +1059.0;
 cc = 9.0;
 x_15 = x_14
    +1052.0;
 bb = 10.0;
 x_16 = x_15
    +1053.0;
 aa = 11.0;
 b_23 = b_22
    +1054.0;
 aa = 12.0;
goto L330;
L330:
 res_054 =
    res_053
    +1063.0;
 aa = 13.0;
 res_055 =
    res_054
    -1062.0;
 aa = 14.0;
 res_33 = res_32
    +1068.0;
 bb = 15.0;
 res_056 =
    res_055
    +1069.0;
 bb = 16.0;
goto L340;
L340:
 y_15 = y_14
    +1070.0;
 dd = 17.0;
 y_16 = y_15
    -1076.0;
 aa = 18.0;
 res_057 =
    res_056
    -1079.0;
 aa = 19.0;
 b_24 = b_23
    +1085.0;
 bb = 20.0;
goto L350;
L350:
 x_17 = x_16
    +1079.0;
 dd = 21.0;
 res_058 =
    res_057
    -1086.0;
 cc = 22.0;
goto L360;
L360:
 res_059 =
    res_058
    -1090.0;
 dd = 23.0;
 num_14 = num_13
    +1095.0;
 bb = 24.0;
 b_25 = b_24
    +1102.0;
 aa = 25.0;
 num_15 = num_14
    +1107.0;
 dd = 26.0;
goto L370;
L370:
 res_060 =
    res_059
    -1108.0;
 cc = 27.0;
 res_061 =
    res_060
    -1098.0;
goto L380;
L380:
 aa = 28.0;
 b_26 = b_25
    +1099.0;
 bb = 29.0;
 res_34 = res_33
    +1100.0;
 bb = 30.0;
 res_062 =
    res_061
    -1101.0;
 cc = 2.0;
 res_063 =
    res_062
    -1037.0;
goto L390;
L390:
 bb = 3.0;
 y_17 = y_16
    -1035.0;
 bb = 4.0;
 res_064 =
    res_063
    -1039.0;
 cc = 5.0;
 c_10 = c_9
    -1042.0;
 cc = 6.0;
 b_27 = b_26
    +1052.0;
goto L400;
L400:
 aa = 7.0;
 num_16 = num_15
    +1053.0;
 cc = 8.0;
 res_065 =
    res_064
    +1065.0;
 cc = 9.0;
 res_066 =
    res_065
    +1068.0;
 cc = 10.0;
 res_35 = res_34
    -1070.0;
goto L410;
L410:
 aa = 11.0;
 x_18 = x_17
    -1066.0;
 cc = 12.0;
 y_18 = y_17
    +1067.0;
 aa = 13.0;
 res_067 =
    res_066
    +1078.0;
 aa = 14.0;
 b_28 = b_27
    +1082.0;
goto L420;
L420:
 aa = 15.0;
 y_19 = y_18
    -1083.0;
 dd = 16.0;
 res_068 =
    res_067
    -1076.0;
 bb = 17.0;
 res_069 =
    res_068
    +1072.0;
 bb = 18.0;
 c_11 = c_10
    -1073.0;
goto L430;
L430:
 aa = 19.0;
 y_20 = y_19
    -1090.0;
 dd = 20.0;
 res_070 =
    res_069
    +1080.0;
 dd = 21.0;
 b_29 = b_28
    +1079.0;
 aa = 22.0;
 c_12 = c_11
    +1091.0;
goto L440;
L440:
 c_13 = c_12
    -1155.0;
 bb = 23.0;
 res_071 =
    res_070
    +1081.0;
 dd = 24.0;
 num_17 = num_16
    +1096.0;
 num_18 = num_17
    -13135.0;
 dd = 25.0;
 res_36 = res_35
    +1090.0;
goto L450;
L450:
 aa = 26.0;
 x_19 = x_18
    +1102.0;
 x_20 = x_19
    +2188.0;
 dd = 27.0;
 res_37 = res_36
    -1091.0;
 cc = 28.0;
 res_38 = res_37
    +1081.0;
 aa = 29.0;
goto L460;
L460:
 b_30 = b_29
    +1082.0;
 b_31 = b_30
    -18779.0;
 aa = 30.0;
 y_21 = y_20
    +1083.0;
 y_22 = y_21
    +4251.0;
 bb = 31.0;
 res_072 =
    res_071
    -1084.0;
 res_073 =
    res_072
    -3932.0;
goto END;
L405925:
 dd = 45.0;
 y_2 = y_1
    -1090.0;
 bb = 46.0;
 res_017 =
    res_016
    +1091.0;
 aa = 47.0;
 b_2 = b_1
    +1092.0;
 dd = 48.0;
 res_16 = res_15
    +1093.0;
goto L70;
L5699997:
 aa = 53.0;
goto L14300003;
L8304993:
 cc = 25.0;
 num_4 = num_3
    +1084.0;
 cc = 26.0;
 res_09 = res_08
    +1085.0;
 aa = 27.0;
 x0 = x+1086.0;
 cc = 28.0;
 num_5 = num_4
    +1076.0;
goto L30;
L8539307:
 c_4 = c_3
    -1102.0;
 dd = 58.0;
 res_019 =
    res_018
    +1103.0;
 bb = 59.0;
 num_8 = num_7
    +1104.0;
 aa = 60.0;
 res_020 =
    res_019
    +1105.0;
 dd = 61.0;
goto L80;
L14300003:
 num_7 = num_6
    -1098.0;
 cc = 54.0;
 x_6 = x_5
    -1099.0;
 bb = 55.0;
 res_18 = res_17
    +1100.0;
 aa = 56.0;
 b_4 = b_3
    +1101.0;
 bb = 57.0;
goto L8539307;
L31243591:
 bb = 13.0;
 y0 = y+1070.0;
 aa = 14.0;
 c0 = c+1067.0;
 aa = 15.0;
 num_1 = num00
    +1074.0;
 cc = 16.0;
 c_1 = c0+1082.0;
goto L0035751343;
L35361786:
 bb = 131.0;
 res_28 = res_27
    +1185.0;
 cc = 132.0;
 res_046 =
    res_045
    +1189.0;
 aa = 133.0;
 res_047 =
    res_046
    -1187.0;
 dd = 134.0;
 res_048 =
    res_047
    +1188.0;
goto L270;
L0035751343:
 aa = 17.0;
 res_12 = res_11
    -1078.0;
 bb = 18.0;
 res_06 = res_05
    -1077.0;
 bb = 19.0;
 num_2 = num_1
    +1078.0;
 bb = 20.0;
 res_07 = res_06
    +1079.0;
goto L20;
L62541504:
 b_7 = b_6
    +1122.0;
 aa = 78.0;
 x_9 = x_8
    +1123.0;
 dd = 79.0;
 res_030 =
    res_029
    +1124.0;
 aa = 80.0;
 res_031 =
    res_030
    -1125.0;
 bb = 81.0;
goto L130;
END:
 res_073=b_31+
    c_13;
 res_073=res_073/
    num_18;
 aa = 32.0;
 res_39 = res_38
    -1085.0;
 res_40 = res_39
    +1972.0;
 res_40=x_20+y_22
    ;
 res_073=res_073*
    res_40;
printf("%d \n",
   res_073);
return 0;
}
```

Appendix C

Licensing for Popper Game Art

Game art reference	Type of license	Owner	Changes made
Player Character (3.7d)	`http://creativecommons.org/licenses/by/3.0/legalcode` `http://www.gnu.org/licenses/gpl-3.0.html` `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`	Curt. *Characters, Zombies and Weapons. Oh My!*, December 26, 2013, OpenGameArt.Org	No changes were made
Balloons (3.7g)	`http://creativecommons.org/publicdomain/zero/1.0/legalcode`	GameArtForge. *Balloons - Set 01*, December 13, 2012, OpenGameArt.Org	The size of the art was reduced and expression symbols removed
Background-Tileset (3.6, 3.7e)	`http://creativecommons.org/licenses/by/3.0/legalcode`	Vicplay. *Grassland Platformer 32x Tileset*, May 31, 2013, OpenGameArt.Org	The size of the art was reduced
Hearts (3.8)	`http://creativecommons.org/licenses/by-sa/3.0/legalcode`	C.Nilsson *Simple small pixel hearts*, June 7, 2012, OpenGameArt.Org	No changes were made
Cactus (3.9)	`http://creativecommons.org/publicdomain/zero/1.0/legalcode`	qubodup *Dirty Railshooter*, November 23, 2010, OpenGameArt.Org	No changes were made
Gems (3.7h)	`http://creativecommons.org/publicdomain/zero/1.0/legalcode`	Paulius Jurgeleviius *Puzzle game jewels*, December 9, 2010, OpenGameArt.Org	No changes were made
Neon Tokens (3.7b)	`http://creativecommons.org/publicdomain/zero/1.0/legalcode`	Naarshakta *Neon style tokens & GUI*, July 15, 2014, OpenGameArt.Org	No changes were made
Ghosts and Monkeys (3.7c, 3.7a)	`http://creativecommons.org/publicdomain/zero/1.0/legalcode`	ryan.dansie *Round Animals*, February 23, 2014, OpenGameArt.Org	No changes were made