Spring 4-28-2018

Behavior-Based Authentication System

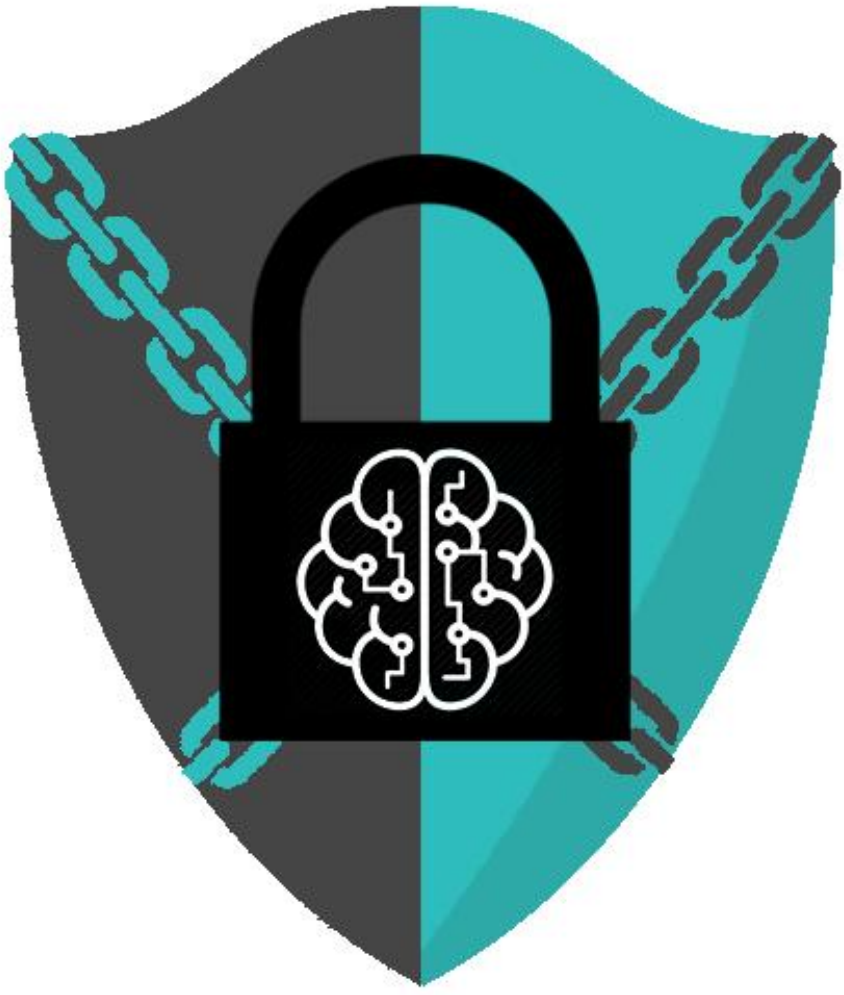
Taylor Means
tmeans1@uwyo.edu

Jared Frank
University of Wyoming, jfrank8@uwyo.edu

Behavior-Based Authentication System

Final Design Document

Jared Frank & Taylor Means



1. Abstract

All current forms of authentication are exploitable via social engineering, theft, hacking, or replication. Due to this, a new form of authentication should be explored: behavioral. A solution to this problem would result in more secure digital environment, including physical access to computers as well as software access. The maze-solving approach presented by this project allows for multiple variables to be observed within a user, presenting many facets of behavior that can be analyzed. In order to solve this problem, enough parameters must be collected and contrasted against one another in order to tell different humans apart from each other based on how they solve a maze.

Other methods of currently existing authentication rely on what you own (physical keys), what you know (passwords), and what you have (biometrics). By creating a randomly generated maze and having an observer AI object keep track of how different users solve a maze, we are able to tell two different users apart from one another to a certain degree of accuracy. Our AI factors in variables such as time spent moving the player, time spent not moving, backtracking, strategy, and more.

2. Overview

This Design Document outlines all of the specifications that are a part of the final Behavior-Based Authentication Project. Its purpose is to summarize the work and allow for replication and expansion of the software, allowing it to continually evolve and become more effective. This document will give an overview of the actual maze application, look into limitations and challenges during the development process, provide a specific breakdown of each class as it is implemented in code, analyze results, and discuss the future of the project.

A maze-solving approach was chosen as it presents a variety of variables that can be analyzed while a user tackles the objective. The design approach for the Behavior-Based Authentication System (BBAS) lies within utilizing the 3rd party Unity game development platform to create a 3-Dimensional randomly generated maze, with code running in the background that analyzes the different ways that users complete the maze. The overall design goals include the ability to differentiate two users (User A and User B) from each other in a consistent fashion. In theory, the two users should be able to solve endless mazes and always be identified as the correct user. Aspects such as time to complete, time moving, and general strategy will be observed.

3. Project Summary

The Program must present an ability to ‘Set’ two Users (A and B) by presenting each individual with 5 different mazes to solve. Solving the maze requires the player to collect two secondary green orbs that can be collected in any order, followed by a final red orb. The orbs are located in the corners of the maze. After individually solving 5 mazes, their data will be stored and the ability to test a single maze will be presented. Upon completion of this single maze, the program will identify which User solved said ‘test maze’. The entire program will utilize the third party Unity3D Engine to render and instantiate all appropriate objects and scripts.

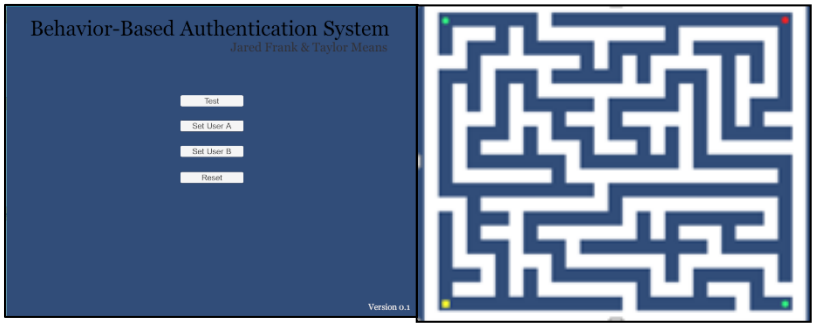


Figure 1: Example of UI and Maze Concept

3a. Tasks

- Random Maze Generation Algorithm paired with appropriate rendered Unity game object assets (walls)
- UI to properly direct setting of Users A and B as well as allow the testing of the actual authentication
- Player and Orb objects that are interfaced with inside the maze, including:
    - Checking that the secondary orbs are collected prior to the final
    - Recognizing when the player collides with an invisible ‘backtrack’ unit
- Observer AI Class that ‘watches’ users play and collect data on (as well as evaluate and compare data through a proprietary algorithm on):
    - Total time to solve the maze
    - Proportion of keys being held down vs. not being held down
    - Proportion of pauses of longer than a specific amount of time vs. total time to solve
    - Proportion of ‘backtracking’ vs. total time to solve
    - Proportion of key ‘bursts’ (rapidly pressing three separate keys back to back) vs. total time to solve
- In addition, the Observer AI Class must interact with the maze by spawning invisible ‘backtrack’ units behind the player in regular time intervals

4. Detailed Design

4a. Structure

There are four total scripts involved in this project, along with a lot of scripting work done through the Unity API that cannot be as easily documented. The scripts are as follows:

- MazeGenerator – To create a randomly generated maze each time it is ran
- menuTest – To keep track of User and Test data
- Observer – To analyze player style and store values/tell them apart
- PlayerController – To allow user interaction with the maze

4b. Maze Generator

The Maze Generator uses a 2-dimensional array to store “Cells” which store a path type—either a path or a wall—and the directions that have already been tried. The 2-dimensional array holds the maze to be drawn into Unity using “wall” prefabs created. The generator will pick a direction at random, if the direction is out of bounds of the array, or there is a path already ahead of the direction, the direction will be marked and remembered, but no path will be laid out. If it passes the conditions, and it is within array bounds and no paths ahead, a path will be laid in two indices straight ahead, and the direction will be marked and remembered. Then, using recursion, the next index in the array will have a random direction chosen and it will start the process over. Once a direction is impossible to be a path, the algorithm will backtrack, and remembering the directions already used, it will try to find another path for the most previous index. Once there is no longer any path to be laid the maze is finished and it is ready to be drawn into Unity.

4c. MenuTest

The menuTest script is relatively simple, with four different functions. It is integrated via the Unity API with the four UI buttons – “test”, “set user A”, “set user B”, and “reset”. The “test” button first checks to make sure it has data for User A and B compiled, then loads up an instance of the Maze scene in the game, communicating with the Observer to let it know that it will be testing the two users against each other. The set user buttons let the Observer know that it’s just setting User data for the respective user. The reset button makes it required for set user A and set user B to be completed again prior to the test button being functional.

4d. Observer

The Observer is the most complex class in the project. During every frame of the game, it collects two different time intervals: total time, and the time a key is held down. In addition, it carefully registers key presses to be able to tell when a ‘burst’ happens, which is registered as three keys pressed in a row within 0.2 seconds of each other. In

addition, it watches the player, and at standard intervals of distance, it spawns a backtrack unit in the opposite direction that the player is travelling, to keep track of backtracks.

Once a player has solved the maze, the observer uses its reportData function, which first checks to see if this is a set user maze or a test maze. If it's a set user maze, it will write down the users data and compile it until all 5 mazes are complete, at which point it averages the data together and writes it to the Unity filewrite system, PlayerPrefs. If it's a test maze, it calls its own evaluate function to distribute points to User A and B based off of the data of the test maze. If User A has more points, they are selected as the user that completed the test maze, and vice versa.

The evaluate function runs as follows on each measured variable:

*Points for User A = (Difference of B / Range of Data) * Weight*

*Points for User B = (Difference of A / Range of Data) * Weight*

Where Difference of = |*Test Variable - Average of User*|

This allows for partial point distribution on each variable being factored in. The 'weight' variable address the fact that the observed metrics can have high variability depending upon its dependence on maze structure. For this reason, some metrics are given a higher weight than others in order to reduce said variability. Based off of testing and some common sense, the weight for different variables are as follows:

- Total time to solve the maze: 1
- Back Track proportion: 1
- Pause proportions: 2
- Key held down proportion: 3
- Key bursts: 3

4e. PlayerController

The PlayerController script handles both user input as well as physics for the player's interaction within the maze. Upon receiving a key input, the player moves in the direction as per the arrow key direction. In addition, it has a Rigidbody Unity physics component that prevents it from clipping through walls, as well as allowing it to register collisions with game objects. The type of game objects it is scripted to collide with include the green orbs (which can be collided with at any time and are destroyed after collision), the red orb (which can only be collided with after both green orbs are destroyed and registers the maze as 'complete'), and backtrack units, which tell the observer that the user has registered a single backtrack unit.

5. Deliverables

5a. Maze Generator

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MazeGenerator : MonoBehaviour
{
    public GameObject wall;
    //public GameObject path; used for testing

    //set up the directions, give them integer values
    private const int UP = 0;
    private const int DOWN = 1;
    private const int LEFT = 2;
    private const int RIGHT = 3;

    //set up the height and width of the maze.
    //they should be alterable in the editor
    public int mazeHeight = 29;
    public int mazeWidth = 29;
    public float camSize = 20f;

    //Used for random number generation
    System.Random rnd = new System.Random();

    //2D array of type Cell will be the "maze"
    Cell[,] maze;

    // Use this for initialization
    void Start()
    {
        Camera.main.orthographic = true;
        Camera.main.orthographicSize = camSize;
        maze = new Cell[mazeHeight, mazeWidth];
        GenerateMaze();
    }

    // Update is called once per frame
    void Update()
    {

    }


    //path will show either a path or a wall, 0 = wall, 1 = path
    //visited will show the directions that have been used so far
    public class Cell
    {
        private int path;
        private int[] visited = new int[4];

        public int getPath()
        {
            return path;
        }

        public void setPath(int a)
        {
            path = a;
        }

        //the index indicates the direction, 1 and 0 indicates used/not
        public void setDir(int a)
        {
```

```
            visited[a] = 1;
        }

        public int getVisited(int a)
        {
            return visited[a];
        }

        //constructor for a Cell
        //create the cell to be a wall
        //mark every direction as not visited
        public Cell()
        {
            path = 0;
            for (int i = 0; i < 4; i++)
            {
                visited[i] = 0;
            }
        }
    };

    public void GenerateMaze()
    {
        InitMaze();
        Draw();
    }

    public void InitMaze()
    {
        //Fill up the 2D array with Cells, maze coordinates
        for (int i = 0; i < mazeHeight; i++)
        {
            for (int j = 0; j < mazeWidth; j++)
            {
                Cell cell = new Cell();
                maze[i, j] = cell;
            }
        }
        //make a path at the beginning of the maze
        maze[mazeHeight - 2, 1].setPath(1);
        mazeDir(mazeHeight - 2, 1);
    }

    public void mazeDir(int height, int width)
    {
        //choose a direction randomly
        //if the direction is impossible back track
        //to a different direction, if no directions are
        //possible back track even further until you are
        //at the "starting" node using recursion.
        for (int z = 0; z < 5; z++)
        {
            int dir = rnd.Next(0, 4);

            for (int i = 0; i < 4; i++)
            {
                   (maze[height, width].getVisited(i) == 1 &&
                    dir == i)
                {
                    bool done = false;
                    while (!done)
                    {
                        int placeholder = rnd.Next(0, 4);
                           (placeholder != dir)
                        {
                            dir = placeholder;
                            done = true;
                        }
                    }
```

```
            }
        }

        (dir == UP)
        {
            // Debug.Log("UP");
            ((height - 2 <= 0) || (maze[height - 2, width].getPath() == 1)) //check if out of bound
s
                                                                            //or if there is a path
 already placed
            {
                //set the direction tried
                maze[height, width].setDir(UP);
                //backtrack
                continue;
            }
                (maze[height, width].getVisited(UP) == 1)
            {
                continue;
            }
                (maze[height - 2, width].getPath() == 0) //check if wall (pass)
            {
                //Change the walls to paths
                maze[height - 1, width].setPath(1);
                maze[height - 2, width].setPath(1);
                //update the direction for the cell
                maze[height, width].setDir(UP);
                //try to make another path using updated location
                mazeDir(height - 2, width);
            }
        }
            (dir == DOWN)
        {
            // Debug.Log("DOWN");
            ((height + 2 >= mazeHeight) || (maze[height + 2, width].getPath() == 1)) //check if out
 of bounds
                                                                                    //or if there
is a path already placed
            {
                //set the direction tried
                maze[height, width].setDir(DOWN);
                //backtrack
                continue;
            }
                (maze[height, width].getVisited(DOWN) == 1)
            {
                continue;
            }
                (maze[height + 2, width].getPath() == 0) //check if wall
            {
                //Change the walls to paths
                maze[height + 1, width].setPath(1);
                maze[height + 2, width].setPath(1);
                //update the direction for the cell
                maze[height, width].setDir(DOWN);
                //try to make another path using the updated location
                mazeDir(height + 2, width);
            }
        }
            (dir == LEFT)
        {
            //Debug.Log("LEFT");
            ((width - 2 <= 0) || (maze[height, width - 2].getPath() == 1)) //check if out of bounds
                                                                            //or if there is a path
already placed
            {
                //set the direction tried
                maze[height, width].setDir(LEFT);
                //backtrack
```

```
                    continue;
                }
                    (maze[height, width].getVisited(LEFT) == 1)
                {
                    continue;
                }
                    (maze[height, width - 2].getPath() == 0) //check if wall
                {
                    //Change the walls to paths
                    maze[height, width - 1].setPath(1);
                    maze[height, width - 2].setPath(1);
                    //update the direction for the cell
                    maze[height, width].setDir(LEFT);
                    //try to make another path using updated location
                    mazeDir(height, width - 2);
                }
            }
                    (dir == RIGHT)
            {
                //Debug.Log("RIGHT");
                   ((width + 2 >= mazeWidth) || (maze[height, width + 2].getPath() == 1)) //check if out o
f bounds
                                                                                          //or if there is
 a path already placed
                {
                    //set the direction tried
                    maze[height, width].setDir(RIGHT);
                    //backtrack
                    continue;
                }
                    (maze[height, width].getVisited(RIGHT) == 1)
                {
                    continue;
                }
                    (maze[height, width + 2].getPath() == 0) //check if wall
                {
                    //Change the walls to paths
                    maze[height, width + 1].setPath(1);
                    maze[height, width + 2].setPath(1);
                    //update the direction for the cell
                    maze[height, width].setDir(RIGHT);
                    //try to make another path using updated location
                    mazeDir(height, width + 2);
                }
            }
        }
    }

    void Draw()
    {
        /*
        //block of code to remove random walls to possibly add
        //a greater variety of paths.
        Cell[,] walls = new Cell[mazeHeight, mazeWidth];

        for (int k = 0; k < 3; k++)
        {
            int randHeight = rnd.Next(mazeHeight / 2 - 5, mazeHeight / 2 + 5);
            int randWidth = rnd.Next(mazeWidth / 2 - 5, mazeWidth / 2 + 5);

            if (maze[randHeight, randWidth].getPath() == 0)
            {
                maze[randHeight, randWidth].setPath(1);
            }
            else
            {
                k--;
            }
        }//Testing*/
```

```
        for (int i = 0; i < mazeHeight; i++)
        {
            for (int j = 0; j < mazeWidth; j++)
            {
                   (maze[i, j].getPath() == 0) //wall
                {
                    Vector3 pos = new Vector3(i, 1, j);
                    Instantiate(wall);
                    wall.transform.position = pos;
                    Debug.Log(maze[i, j].getPath());
                }
               // else //path
               // {
                    //Vector3 pos = new Vector3(i, 1, j);
                    //Instantiate(path);
                    //path.transform.position = pos;
                    //Debug.Log(maze[i, j].getPath());
                //}
                //Testing purposes


            }
        }
    }
}
```

5b. MenuTest

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class menuTest : MonoBehaviour {

   // FileHandler f1 = new FileHandler();

    public void nextScene()
    {
           ((PlayerPrefs.GetInt ("ARegistered") == 1) && (PlayerPrefs.GetInt ("BRegistered") == 1)) {
            PlayerPrefs.SetInt ("testType", 0);
            PlayerPrefs.SetInt ("repetitions", 0);
            SceneManager.LoadScene ("maze");
        }
    }

    public void nextSceneA()
    {
        PlayerPrefs.SetInt ("testType", 1);
        PlayerPrefs.SetInt ("repetitions", 0);
        SceneManager.LoadScene("maze");
    }

    public void nextSceneB()
    {
        PlayerPrefs.SetInt ("testType", 2);
        PlayerPrefs.SetInt ("repetitions", 0);
        SceneManager.LoadScene("maze");
    }

    public void reset()
    {
        PlayerPrefs.SetInt ("ARegistered", 0);
        PlayerPrefs.SetInt("BRegistered", 0);
    }

}
```

5c. Observer

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
//using UnityEditor;
using System.IO;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class Observer : MonoBehaviour
{
    private float totalTime;
    private float keyHeldTime;
    private float backtrackUnits;
    //private float noKeyTime;
    private float aPoints;
    private float bPoints;
    private float lastSpawn;
    private float burstCount;
    private float bursts;
    private float burstTime;
    private float pauses;
    private float pauseCount;
    private float pauseTime;
    private float tempCount;
    private bool burstRegistered;
    public TextAsset testA;
    public TextAsset testB;
    public TextAsset master;
    public TextAsset test;
    public Text userARecognized;
    public Text userBRecognized;
    public GameObject player;
    public GameObject backtracker;

    // Use this for initialization
    void Start()
    {
        userARecognized.enabled = false;
        userBRecognized.enabled = false;
        aPoints = 0;
        bPoints = 0;
        backtrackUnits = 0;
        lastSpawn = 0;
        bursts = 0;
        burstCount = 0;
        burstTime = 0;
        pauses = 0;
        pauseCount = 0;
        tempCount = 0;
        pauseTime = 0;
        burstRegistered = false;
    }

    // Update is called once per frame
    void Update()
    {
        totalTime += Time.deltaTime;

           (Input.anyKey)
            keyHeldTime += Time.deltaTime;

           (keyHeldTime > 3 + lastSpawn)
        {
            lastSpawn += 3;
               (Input.GetKey(KeyCode.UpArrow))
```

```
                spawnBacktracker(1);
                    (Input.GetKey(KeyCode.DownArrow))
                spawnBacktracker(2);
                    (Input.GetKey(KeyCode.LeftArrow))
                spawnBacktracker(3);
                    (Input.GetKey(KeyCode.RightArrow))
                spawnBacktracker(4);
        }

        //else
        //noKeyTime += Time.deltaTime;
        // code for burst
           (Input.GetKeyUp(KeyCode.UpArrow) || Input.GetKeyUp(KeyCode.DownArrow) || Input.GetKeyUp(KeyCode
.LeftArrow) || Input.GetKeyUp(KeyCode.RightArrow))
        {
            pauseCount++;
            burstCount++;
            burstTime = 0;
            burstRegistered = true;
               (burstCount > 2)
            {
                burstCount = 0;
                bursts++;
            }
        }

           (burstRegistered)
        {
            burstTime += Time.deltaTime;
               (burstTime > 0.2f)
            {
                burstRegistered = false;
                burstCount = 0;
            }
        }

           (pauseCount > 0)
        {
              (!Input.anyKeyDown)
            {
                pauseTime += Time.deltaTime;
                   (pauseTime > 2.0f)
                {
                      (pauseCount != tempCount)
                    {
                       pauses++;
                    }
                    tempCount = pauseCount;
                }
            }

            {
                pauseTime = 0;
            }
        }
    }

    IEnumerator pause()
    {
              return new WaitForSecondsRealtime(5);
        SceneManager.LoadScene("menu");
    }

    private void spawnBacktracker(int key)
    {
              (key)
        {
                 1:
                Vector3 movement = new Vector3(0f, 0f, -1f);
```

```
                var bt = (GameObject)Instantiate(backtracker, player.transform.position + movement, player
.transform.rotation);
                break;
                 2:
                movement = new Vector3(0f, 0f, 1f);
                bt = (GameObject)Instantiate(backtracker, player.transform.position + movement, player.tra
nsform.rotation);
                break;
                 3:
                movement = new Vector3(1f, 0f, 0f);
                bt = (GameObject)Instantiate(backtracker, player.transform.position + movement, player.tra
nsform.rotation);
                break;
                 4:
                movement = new Vector3(-1f, 0f, 0f);
                bt = (GameObject)Instantiate(backtracker, player.transform.position + movement, player.tra
nsform.rotation);
                break;
                   :
                bt = (GameObject)Instantiate(backtracker, player.transform.position, player.transform.rota
tion);
                break;
        }
    }

    public void incrementBacktrack()
    {
        backtrackUnits++;
    }

    public void reportData(int type, int r)
    {
        // testing two users
        Debug.Log(type);
           (type == 0)
        {
            float ATotalTime = (PlayerPrefs.GetFloat("ATotalTime1") + PlayerPrefs.GetFloat("ATotalTime2")
+ PlayerPrefs.GetFloat("ATotalTime3") + PlayerPrefs.GetFloat("ATotalTime4") + PlayerPrefs.GetFloat("ATotal
Time5")) / 5;
            float AKeyHeldProp = (PlayerPrefs.GetFloat("AKeyHeldProp1") + PlayerPrefs.GetFloat("AKeyHeldPr
op2") + PlayerPrefs.GetFloat("AKeyHeldProp3") + PlayerPrefs.GetFloat("AKeyHeldProp4") + PlayerPrefs.GetFlo
at("AKeyHeldProp5")) / 5;
            float ABacktrackProp = (PlayerPrefs.GetFloat("ABacktrackProp1") + PlayerPrefs.GetFloat("ABackt
rackProp2") + PlayerPrefs.GetFloat("ABacktrackProp3") + PlayerPrefs.GetFloat("ABacktrackProp4") + PlayerPr
efs.GetFloat("ABacktrackProp5")) / 5;
            float ABurstProp = (PlayerPrefs.GetFloat("ABurstProp1") + PlayerPrefs.GetFloat("ABurstProp2")
+ PlayerPrefs.GetFloat("ABurstProp3") + PlayerPrefs.GetFloat("ABurstProp4") + PlayerPrefs.GetFloat("ABurst
Prop5")) / 5;
            float APauses = (PlayerPrefs.GetFloat("APauses1") + PlayerPrefs.GetFloat("APauses2") + PlayerP
refs.GetFloat("APauses3") + PlayerPrefs.GetFloat("APauses4") + PlayerPrefs.GetFloat("APauses5")) / 5;

            float BTotalTime = (PlayerPrefs.GetFloat("BTotalTime1") + PlayerPrefs.GetFloat("BTotalTime2")
+ PlayerPrefs.GetFloat("BTotalTime3") + PlayerPrefs.GetFloat("BTotalTime4") + PlayerPrefs.GetFloat("BTotal
Time5")) / 5;
            float BKeyHeldProp = (PlayerPrefs.GetFloat("BKeyHeldProp1") + PlayerPrefs.GetFloat("BKeyHeldPr
op2") + PlayerPrefs.GetFloat("BKeyHeldProp3") + PlayerPrefs.GetFloat("BKeyHeldProp4") + PlayerPrefs.GetFlo
at("BKeyHeldProp5")) / 5;
            float BBacktrackProp = (PlayerPrefs.GetFloat("BBacktrackProp1") + PlayerPrefs.GetFloat("BBackt
rackProp2") + PlayerPrefs.GetFloat("BBacktrackProp3") + PlayerPrefs.GetFloat("BBacktrackProp4") + PlayerPr
efs.GetFloat("BBacktrackProp5")) / 5;
            float BBurstProp = (PlayerPrefs.GetFloat("BBurstProp1") + PlayerPrefs.GetFloat("BBurstProp2")
+ PlayerPrefs.GetFloat("BBurstProp3") + PlayerPrefs.GetFloat("BBurstProp4") + PlayerPrefs.GetFloat("BBurst
Prop5")) / 5;
            float BPauses = (PlayerPrefs.GetFloat("BPauses1") + PlayerPrefs.GetFloat("BPauses2") + PlayerP
refs.GetFloat("BPauses3") + PlayerPrefs.GetFloat("BPauses4") + PlayerPrefs.GetFloat("BPauses5")) / 5;

            evaluate(totalTime, ATotalTime, BTotalTime, 1);
            evaluate(keyHeldTime / totalTime, AKeyHeldProp, BKeyHeldProp, 3);
            evaluate(backtrackUnits / totalTime, ABacktrackProp, BBacktrackProp, 1);
```

```
    evaluate(bursts / totalTime, ABurstProp, BBurstProp, 3);
    evaluate(pauses, APauses, BPauses, 2);

       (aPoints > bPoints)
    {
        userARecognized.enabled = true;
    }

    {
        userBRecognized.enabled = true;
    }

    // wait for seconds before sending back to menu
    StartCoroutine(pause());

}
// setting user A
        (type == 1)
{
          (r)
    {
             1:
            PlayerPrefs.SetFloat("ATotalTime1", totalTime);
            PlayerPrefs.SetFloat("AKeyHeldProp1", keyHeldTime / totalTime);
            PlayerPrefs.SetFloat("ABacktrackProp1", backtrackUnits / totalTime);
            PlayerPrefs.SetFloat("ABurstProp1", bursts / totalTime);
            PlayerPrefs.SetFloat("APauses1", pauses / totalTime);
            SceneManager.LoadScene("maze");
            break;
             2:
            PlayerPrefs.SetFloat("ATotalTime2", totalTime);
            PlayerPrefs.SetFloat("AKeyHeldProp2", keyHeldTime / totalTime);
            PlayerPrefs.SetFloat("ABacktrackProp2", backtrackUnits / totalTime);
            PlayerPrefs.SetFloat("ABurstProp2", bursts / totalTime);
            PlayerPrefs.SetFloat("APauses2", pauses / totalTime);
            SceneManager.LoadScene("maze");
            break;
             3:
            PlayerPrefs.SetFloat("ATotalTime3", totalTime);
            PlayerPrefs.SetFloat("AKeyHeldProp3", keyHeldTime / totalTime);
            PlayerPrefs.SetFloat("ABacktrackProp3", backtrackUnits / totalTime);
            PlayerPrefs.SetFloat("ABurstProp3", bursts / totalTime);
            PlayerPrefs.SetFloat("APauses3", pauses / totalTime);
            SceneManager.LoadScene("maze");
            break;
             4:
            PlayerPrefs.SetFloat("ATotalTime4", totalTime);
            PlayerPrefs.SetFloat("AKeyHeldProp4", keyHeldTime / totalTime);
            PlayerPrefs.SetFloat("ABacktrackProp4", backtrackUnits / totalTime);
            PlayerPrefs.SetFloat("ABurstProp4", bursts / totalTime);
            PlayerPrefs.SetFloat("APauses4", pauses / totalTime);
            SceneManager.LoadScene("maze");
            break;
             5:
            PlayerPrefs.SetFloat("ATotalTime5", totalTime);
            PlayerPrefs.SetFloat("AKeyHeldProp5", keyHeldTime / totalTime);
            PlayerPrefs.SetFloat("ABacktrackProp5", backtrackUnits / totalTime);
            PlayerPrefs.SetFloat("ABurstProp5", bursts / totalTime);
            PlayerPrefs.SetFloat("APauses5", pauses / totalTime);
            PlayerPrefs.SetInt("ARegistered", 1);
            SceneManager.LoadScene("menu");
            break;
               :
            break;
    }

}
// setting user B
        (type == 2)
```

```
        {
                (r)
            {
                 1:
                PlayerPrefs.SetFloat("BTotalTime1", totalTime);
                PlayerPrefs.SetFloat("BKeyHeldProp1", keyHeldTime / totalTime);
                PlayerPrefs.SetFloat("BBacktrackProp1", backtrackUnits / totalTime);
                PlayerPrefs.SetFloat("BBurstProp1", bursts / totalTime);
                PlayerPrefs.SetFloat("BPauses1", pauses / totalTime);
                SceneManager.LoadScene("maze");
                break;
                 2:
                PlayerPrefs.SetFloat("BTotalTime2", totalTime);
                PlayerPrefs.SetFloat("BKeyHeldProp2", keyHeldTime / totalTime);
                PlayerPrefs.SetFloat("BBacktrackProp2", backtrackUnits / totalTime);
                PlayerPrefs.SetFloat("BBurstProp2", bursts / totalTime);
                PlayerPrefs.SetFloat("BPauses2", pauses / totalTime);
                SceneManager.LoadScene("maze");
                break;
                 3:
                PlayerPrefs.SetFloat("BTotalTime3", totalTime);
                PlayerPrefs.SetFloat("BKeyHeldProp3", keyHeldTime / totalTime);
                PlayerPrefs.SetFloat("BBacktrackProp3", backtrackUnits / totalTime);
                PlayerPrefs.SetFloat("BBurstProp3", bursts / totalTime);
                PlayerPrefs.SetFloat("BPauses3", pauses / totalTime);
                SceneManager.LoadScene("maze");
                break;
                 4:
                PlayerPrefs.SetFloat("BTotalTime4", totalTime);
                PlayerPrefs.SetFloat("BKeyHeldProp4", keyHeldTime / totalTime);
                PlayerPrefs.SetFloat("BBacktrackProp4", backtrackUnits / totalTime);
                PlayerPrefs.SetFloat("BBurstProp4", bursts / totalTime);
                PlayerPrefs.SetFloat("BPauses4", pauses / totalTime);
                SceneManager.LoadScene("maze");
                break;
                 5:
                PlayerPrefs.SetFloat("BTotalTime5", totalTime);
                PlayerPrefs.SetFloat("BKeyHeldProp5", keyHeldTime / totalTime);
                PlayerPrefs.SetFloat("BBacktrackProp5", backtrackUnits / totalTime);
                PlayerPrefs.SetFloat("BBurstProp5", bursts / totalTime);
                PlayerPrefs.SetFloat("BPauses5", pauses / totalTime);
                PlayerPrefs.SetInt("BRegistered", 1);
                SceneManager.LoadScene("menu");
                break;
            }
        }


        // FILE WRITER - UNNECESSARY
        //StreamWriter sw = new StreamWriter ("Assets/UserData/test.txt");
        //sw.WriteLine (totalTime);
        // writing proportion of keyHeldTime
        //sw.WriteLine (keyHeldTime/totalTime);
        //sw.WriteLine (noKeyTime);
        //sw.Close ();
        //AssetDatabase.ImportAsset ("Assets/UserData/test.txt");

        Debug.Log("Total time: " + totalTime + " keyHeldTime: " + keyHeldTime + " backtracks: " + backtrac
kUnits + " bursts: " + bursts + " pauses: " + pauses);
    }

    private void evaluate(float variable, float a, float b, float weight)
    {
        float da = Mathf.Abs(variable - a);
        float db = Mathf.Abs(variable - b);
        float dRange = Mathf.Abs(da + db);

        aPoints += (db / dRange) * weight;
        bPoints += (da / dRange) * weight;
```

```
    }
}
```

5d. PlayerController

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;

public class PlayerController : MonoBehaviour {

    public float speed = 3.0f;
    public Text winText;
    public int count;
    public bool done;
    public Observer obs;
    private Rigidbody rb;
    private int prereq = 2;

    private int testType;
    private int repetitions;

    void Start () {
        testType = PlayerPrefs.GetInt ("testType");
        repetitions = PlayerPrefs.GetInt ("repetitions");
        rb = GetComponent<Rigidbody>();
    }
    /*
    float moveHorizontal = Input.GetAxis("Horizontal");
    float moveVertical = Input.GetAxis("Vertical");
    Vector2 movement = new Vector3(moveHorizontal, 0.1f, moveHorizontal);
    rb.AddForce(movement* speed);
    */
    // Update is called once per frame
    void Update () {
        //Move the character when the key is pressed.
           (Input.GetKey(KeyCode.RightArrow))
        {
            Vector3 position = this.transform.position;
            position.x = position.x + .1f;
            this.transform.position = position;
        }

           (Input.GetKey(KeyCode.LeftArrow))
        {
            Vector3 position = this.transform.position;
            position.x = position.x - .1f;
            this.transform.position = position;
        }

           (Input.GetKey(KeyCode.UpArrow))
        {

            Vector3 position = this.transform.position;
            position.z = position.z + .1f;
            this.transform.position = position;

        }

           (Input.GetKey(KeyCode.DownArrow))
        {
            Vector3 position = this.transform.position;
            position.z = position.z - .1f;
            this.transform.position = position;
        }
    }
```

```
    void OnTriggerEnter (Collider col)
    {
           (col.gameObject.tag == "Finish") {
               (prereq < 1) {
                // put code here to finish game
                repetitions++;
                PlayerPrefs.SetInt ("repetitions", repetitions);
                obs.reportData (testType, repetitions);
                Destroy (this.gameObject);
            }
        }

                (col.gameObject.tag == "Backtrack") {
            obs.incrementBacktrack ();
            Destroy (col.GetComponent<Collider> ().gameObject);
        }

                (col.gameObject.tag == "prereq") {
            prereq--;
            Destroy (col.GetComponent<Collider> ().gameObject);
        }
    }

}
```

6. Conclusion

6a. Limitations & Challenges

The final maze authentication program presented in this design document is limited to just differentiating two users apart, and not with a 100% success rate (see results). Major challenges included coming up with effective metrics for the Observer to account for, particularly without advising from those with behavioral psychology knowledge. The maze generation algorithm itself is also limited to long windy paths, often lacking shortcuts that would give users more choice and thus more of a chance to display behavioral differences from one another. The maze structure lends itself to improper implementation of analyzing backtracking, as often times large amounts of backtracking is unavoidable, as it is more dependent upon the random maze structure rather than behavior. Individuals tend to change behavior sporadically, or as their emotions change. This limits the amount of accurate data the Observer can collect.

6b. Testing & Results

Four different pairs of people were tested against each other as users A and B. Each person performed 5 ‘test mazes’ after setting themselves as a user. A correct identification of the user counts as a success. Success percentiles ranged from a low of 40% to a high of 100%, with an average of a 72.5% success rate. Sample size was low as the testing was largely to observe how this program worked in the hands of users, as well as which metrics were more important for observation.

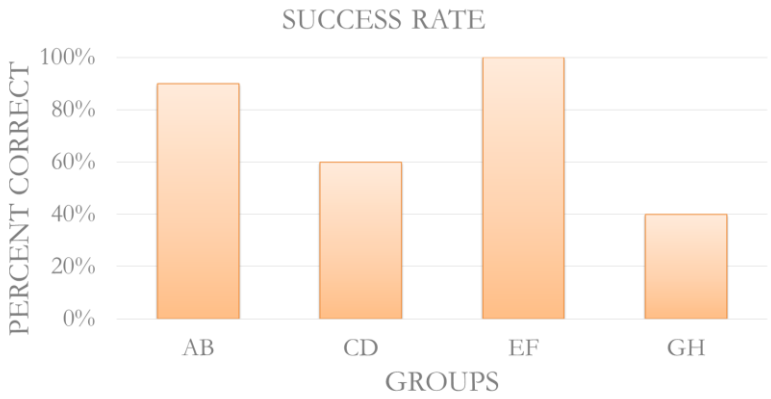


Figure 2: Success Rate Graph

4c. Discussion

Overall, our version of this maze solving program works as a good first step as a proof-of-concept/framework for a theoretical Behavior-Based Authentication System. Clearly, without a 100% success rate for a large sample size, it would not be suitable to serve as a reliable or secure system of authentication as it stands right now. To take steps towards achieving this goal, future work should be performed in conjunction with the consultation of psychologists in order to further delve into what behavioral patterns

expressed on a computer make individuals unique from one another. Continually adding new, effective metrics to be measured will only increase the success rate of the system.

6d. Future Work

- Add new metrics to be collected by the Observer and tweak existing ones until the ability to differentiate two different users is successful at a 100% level
- Expand the efficacy of the program to be able to tell an arbitrarily large user base apart from each other
- Shrink the maze while maintaining the success rate (in theory by continuing to improve the Observer) until it is quick and easy enough to be solved so that it could realistically act as a convenient alternative to passwords, pin numbers, keys, or biometric scanners