Android Malware: De-Smartering the Smart Phone

James Trussel and Seth Martin
Computing and Technology Department, Cameron University, Lawton, OK, USA

Abstract – Mobile malware is relatively new, but growing more prevalent and powerful. It is important to understand the effects of malware on a mobile device in order to combat it. A reason mobile malware has become such a problem recently, is because people have often felt that their mobile devices are immune to infection. This research will attempt to exploit the vulnerabilities of the Android Operating System and the marketplace by creating a virus and testing permissions to determine overall and potential impact. This paper defines and documents the process, identifies problems encountered, and explains the overall findings.

Keywords
Android, malware, mobile devices, security

1. Introduction

The market for mobile devices is growing rapidly [1] [2] and there seems to be no wrong platform to choose; however, they are different and some might be better suited than others for specific needs. Businesses are realizing how powerful mobile devices are and are willing to spend a little extra money in order to increase their efficiency. With more businesses shifting to mobile devices to conduct their business, there has also been a significant increase in the number of security incidents. [2] Because Android OS is open-source and so widely distributed, it is a very likely target for malware. [3] The sheer nature of a single marketplace that is readily available to millions of people at any time of day makes it a good target for malware developers. [4] Since malware measures its success by how many people it can infect, and there are millions of Android users, the Android marketplace makes for ideal prey.

There are different types of malware to which mobile devices are vulnerable. From the generally categorized malware, to spyware or grayware, this paper will define these different threat types, what they are capable of and define ways to combat these threats.

The backbone of the Android Operating System’s security lies with an extensive permissions system which can grant an application access to a variety of controls. [5] Every application requests permission to install and declares the permissions that are needed in order for the application to run properly, but like a Jedi mind trick, the permissions are ignored and users install anyway. [6] This paper discusses those permissions and the actual abilities that are granted when an application is allowed.

The 2.3 version of the Android Operating System, otherwise known as Gingerbread, was the first targeted by mobile malware and will be the subject of tests throughout this paper. The purpose of this paper is to define how the Android OS works, exploit vulnerabilities in the Android OS, document tests and findings, and provide explanations in order to help gain a better understanding of the dangers that mobile devices (specifically Android OS devices) can harbor for businesses.

2. Previous Work

There has been a lot of work and research done regarding topics similar to the topic of mobile malware, many encompassing a broad range of different platforms or devices. Much of the findings have been extremely helpful in researching the topic of Android malware.
Both Google and Apple use an application’s permissions and a review process in order to protect users from malware. A previous study evaluated the effectiveness of permissions and application review in Nokia, Apple and Google devices. [5] The chart below is documented in the paper and displays findings. Because malware detection is left in the hands of the companies that develop the devices, it’s important to understand the security settings on that device and how they can be manipulated to better protect the user from malware.

<table>
<thead>
<tr>
<th>Number of Dangerous permissions</th>
<th>Number of non-malicious applications</th>
<th>Number of malicious applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>75 (8%)</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>154 (16%)</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>182 (19%)</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>152 (16%)</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>140 (15%)</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>82 (9%)</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>65 (7%)</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>28 (3%)</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>10 (2%)</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>21 (2%)</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>10 (1%)</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>6 (0.6%)</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>7 (0.7%)</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>4 (0.4%)</td>
<td>-</td>
</tr>
<tr>
<td>14</td>
<td>4 (0.4%)</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>2 (0.2%)</td>
<td>-</td>
</tr>
<tr>
<td>16</td>
<td>1 (0.1%)</td>
<td>-</td>
</tr>
<tr>
<td>17</td>
<td>1 (0.1%)</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>20</td>
<td>1 (0.1%)</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1 (0.1%)</td>
<td>-</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>26</td>
<td>1 (0.1%)</td>
<td>-</td>
</tr>
</tbody>
</table>

This table is the results of a study conducted testing a sample of 11 pieces of malware and 956 non-malicious applications which may or may not request a specific number of “Dangerous” permissions. Dangerous permissions are those that allow an application to partake in malicious activity. There are numerous activities that can be harmful when given the right permissions. From this table we can see that every malicious application that was tested required at least 1 granted permission and that none of their tested malicious applications required more than 11 granted permissions. It is also shown in this table that even non-malicious applications request some potentially dangerous permissions. [5] A permission is deemed potentially dangerous when, if granted, the ability given by that permission can cause harm to the device, the user’s personally identifiable information can be stolen, or the device can be controlled remotely without the user’s knowledge. What we don’t see is which permission is the most dangerous or which permission is the one permission that is needed in order to perform malicious activities. That is something this paper will determine.

3. Types of Mobile Malware

Malware can be classified in various ways and given various labels; however, for the purposes of this paper, the malware classifications will be General Malware, Personal Spyware, and Grayware. [5] The reason for using these classifications is because they all use mechanisms and have the most concise motivations.

**General Malware.** This type of Malware is constructed for the purpose of gaining access to a device in order to damage it, steal data or to be an annoyance. This type is the most threatening and is considered illegal in most countries. Usually the user is defrauded into installing the malware or access is granted through permissions or through vulnerabilities in the device.

**Personal Spyware.** Like most Spyware, this type collects personal information from your device and sends it to a predetermined location without the device owner’s knowledge, to be used for any purpose the owner of the spyware wants. The sale of spyware is not considered illegal; however, it may be illegal to install spyware without the device owner’s authorization.

**Grayware.** Usually legitimate applications will collect user data and send to a remote location to be used in marketing/data mining or to assist in profiling its users. Grayware is not intended to harm the user, and some parts of the grayware may be used to help the user. The cost for this service; however, is that the device is open to the application for its collection of data. While some users do not like the thought that their personal data is being
used to turn a profit, others do not mind or feel that there isn’t enough risk involved to miss out on the service that grayware provides.

4. Creating Android Applications

In order to create an Android marketplace developer’s account, you need to pay $25 and developers will need to “agree” with the terms and conditions. Once that is done, the account is ready for uploading applications. When creating an Android application, one needs a computer with Eclipse, Android SDK and the Android Development Tools (plugin for Eclipse).

Applications must also meet a few requirements in order to be published to the marketplace, which mostly involves how the marketplace catalogues and encrypts the program. Each application is submitted with an icon and a couple of screenshots. For the purpose of the research done for this paper, Eclipse, Eclipse DDMS and Eclipse AVD were used to develop the test applications.

Eclipse. This is an open-source integrated development environment (IDE) provided by The Eclipse Foundation. Add-ons and plug-ins such as the Android SDK, the Android Virtual Device (AVD) and the Dalvik Debug Monitor Server (DDMS) are designed to be used in conjunction with the Eclipse IDE in order to develop applications for the Android platform. Eclipse is the primary IDE used in Android development and can be used to develop applications in many languages. The Eclipse DDMS is an add-on that is used to monitor and debug applications and devices as well as simulate phone calls, SMS messaging, and simulate or manipulate data transfer rates. The DDMS also displays the running processes and the stack. The Android SDK contains an Android Virtual Device that acts as a mobile device emulator and integrates with Eclipse. The AVD can simulate most of what a real phone can do such as touch screen clicks, phone calls, SMS messages and many other functions. An AVD can be configured with any version of Android and numerous hardware settings.

Applications. For the purpose of this paper, we are developing two applications in order to test Android permissions and explore the capabilities of Android virus. The first app is an Android Activity that allows the users to play a game. With this game we will investigate the limits of various permissions by attempting to capture private data. We will also attempt to reproduce known malware techniques. The second application is an Android Service and Broadcast Receiver that provides the user with a music player. An Android Service is a long running application that can keep running in the background as other applications or used. An Android Broadcast Receiver can receive data send between applications such as phone status updates. We will use the music player to in an attempt to capture private data without the user’s knowledge and reproduce known malware techniques.

Bouncer. Until recently, there was no quality control or approval system in place to validate or check the legitimacy of applications that are on the Android marketplace. Sometime mid-2011, Google added a new layer of security to its application development market and called it Bouncer. The new security feature is fully automated and
scans uploaded applications for known malware and looks for behaviors that might indicate that the application is not behaving as it should, comparing it against previously analyzed applications. While Bouncer has been fairly effective [7], showing a 40 percent decline in the number of downloads of potentially malicious applications, developers have found workarounds. There is a new malware that spreads through Facebook’s mobile application called “any_name.apk” or “allnew.apk” which disguises itself as a seemingly harmless friend request, where the user is redirected and the harmful download begins, completely bypassing Google Bouncer. [8]

5. Security Features

Permissions. Applications that are downloaded will often pose the question, “Allow” the application access to various phone features? Allowing the application use of the Internet is actually allowing it to do a wide variety of tasks like accessing the devices networking hardware settings including Bluetooth, or communicating with an external server without the user’s knowledge. [4]

Sandboxing. There is a virtual wall around applications that naturally keeps them from interacting with one another unless they are designed that way. When an application is written that is designed to interact with another application, this raises suspicion regardless of the legitimacy of the application. Generally, applications that are designed to work together require a manifest with accepted signatures/certificates which are checked before the application allows the interaction to take place. In the Google Play market, formerly known as the Android Marketplace, if the signatures are not identical, suspicion is raised. When an application is uploaded which is scanned and raises suspicion, it is then tested in a virtual environment to determine how it behaves. If the application behaves like malware, it will be denied visibility on the marketplace by Google’s security features, including Bouncer.

Permissions. The Android security setup relies heavily on permissions. Permissions are granted or denied upon downloading an application from the Android Market. Because the permissions are granted by the user, this results in a somewhat ineffective and coarse-grained security system. By nature, users are vulnerabilities to any device. Applications are often downloaded without the knowledge of any suspicious or malicious permissions. Most permissions are bulked into labels or groups when presented to the user. For example, “Allow Internet” could be a label for a permission that not only grants network access but also Bluetooth, Wi-Fi, network status, and any networking communication available to the phone. An “Allow System” permission not only grants access to settings, but may also allow changes to those settings. Some permissions are potentially more dangerous than others when they are broken down into what the permissions advertise actually allow, whether the application is legitimate or malicious. There are many legitimate applications that, if altered, could cause tremendous problems for device users. In fact, these “bogus apps” posing as legitimate applications have been the primary vehicle for malware. Users think they are getting something for free, yet those free applications could cost them their privacy.

TABLE OF DANGEROUS PERMISSIONS WITH GROUPING LABELS AND ACTUAL PERMISSIONS GRANTED [8]

<table>
<thead>
<tr>
<th>Permission Group</th>
<th>Actual Access Granted</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNTS</td>
<td>Permissions for direct access to the accounts managed by the Account Manager.</td>
</tr>
<tr>
<td>COST_MONEY</td>
<td>Used for permissions that can be used to make the user spend money without their direct involvement.</td>
</tr>
<tr>
<td>DEVELOPMENT_TOOLS</td>
<td>Group of permissions that are related to development features.</td>
</tr>
<tr>
<td>HARDWARE_CONTROLS</td>
<td>Used for permissions that provide direct access to the hardware on the device.</td>
</tr>
<tr>
<td>LOCATION</td>
<td>Used for permissions that allow access to the user’s current location.</td>
</tr>
</tbody>
</table>
MESSAGES Used for permissions that allow an application to send messages on behalf of the user or intercept messages being received by the user.

NETWORK Used for permissions that provide access to the hardware on the device.

PERSONAL_INFO Used for permissions that provide access to the user’s private data, such as contacts, calendar events, e-mail messages, etc.

PHONE_CALLS Used for permissions that are associated with accessing and modifying telephony state: intercepting outgoing calls, reading and modifying the phone state.

STORAGE Group of permissions that are related to SD card access.

SYSTEM_TOOLS Group of permissions that are related to system APIs.

Making an android virus begins first with discovering new vulnerabilities or exploiting known vulnerabilities. Once those vulnerabilities are identified, the type of malware must be chosen. Simple malware can gather and send information behind the disguise of another application. Complex malware can instruct the device to text, make purchases, or completely control the device. Understanding the process of creating mobile apps and what to use them for can be difficult enough. Throw in the desire to create malware and a new doorway opens to new possibilities.

6. Testing Permissions

The following data describes tests conducted with the permissions tested, the testing application, and the test procedure with results. Each test procedure was designed for these research purposes to exploit a specific set of instructions and simulate potential malicious activity. The tests were conducted by running the application in a virtual machine. Once the test passes, it is verified, tested again, verified and then documented. If a test fails, it is tested again and verified, code checked, tested again, verified and then documented.

<table>
<thead>
<tr>
<th>Android Permissions Granted</th>
<th>Android Application Used</th>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE_SMS, SEND_SMS, RECEIVE_SMS</td>
<td>snake</td>
<td>Reading/Transmitting SMS Messages to another phone number</td>
<td>Pass – Able to Read and re-transmit SMS messages.</td>
</tr>
<tr>
<td>INTERNET</td>
<td>snake</td>
<td>Connect to remote server and transmit data</td>
<td>Pass – Able to connect and transmit data. Not able to transmit sensitive information.</td>
</tr>
<tr>
<td>INTERNET, BLUETOOTH, BLUETOOTH_ADMIN</td>
<td>snake</td>
<td>Repeatedly call the Bluetooth connection while app is running.</td>
<td>Pass – Able to repeatedly attempt Bluetooth connection on the AVD. Need to test on real device.</td>
</tr>
<tr>
<td>CAMERA</td>
<td>snake2</td>
<td>Take Pictures</td>
<td>Pass – Able to take pictures at random.</td>
</tr>
<tr>
<td>GET_ACCOUNTS, READ_CONTACTS, WRITE_CONTACTS</td>
<td>net.trussel.sample</td>
<td>Read/Write Contact Information and gain access to external mail services.</td>
<td>Pass – Able to read/write contact information only with permission.</td>
</tr>
<tr>
<td>READ_LOGS</td>
<td>N/A</td>
<td>Access Android log file from an application.</td>
<td>Failed – Only able to access Logcat from the IDE</td>
</tr>
</tbody>
</table>

Through testing the permissions shown above, there was a common trend in the results; without permission, access to the desired functions were not possible. Android’s security features are highly dependent on the integrity of its permission.
7. Conclusions

Without the permission of the user, there isn’t much that can be accomplished in the way of malware. While there are some forms of malware that exploit specific operating system vulnerabilities, they still rely on the permissions granted upon the user performing the initial install. Through this research, we have found that all malware discovered is some form of Spyware, Greyware or general Malware. None of the malicious applications researched had virus behaviors (specifically, self-replicating). The majority of these types of applications are used to gather personal data, become an annoyance and possibly cost the user or company money. In our research, we found that many sources disclose vulnerabilities in the operating system, while we view the source of the vulnerability as the user as do many organizations. Corporations spend valuable resources training and keeping their users up to date with the most current threats and prevention systems or software, all for good reason. While experienced and knowledgeable users may not be as susceptible to Android malware, many users that the Android devices’ costs naturally attract are more at risk.

The topic of Android malware is growing even as this paper is written. For example, it has been discovered that there are applications that do not request any permissions, yet are allowed access to SD card data (which is some old news to Google since they tell customers this), handset data, and can see what files are stored by other applications. This was discovered early April, 2012. According to Google, users can avoid theft of their sensitive data by not storing it on their SD card and keeping their phone up to date. As for the purpose of this paper, we would have to concur with Google. Especially in a business environment, it’s important to firmly secure sensitive information and a mobile device is not yet the best place for that. While there are further developments in mobile security, it’s still a very young concept and needs further development.

Though we didn’t reach our final goal and exploit the Operating System, we were able to develop Android malware (APPENDIX A) and use it to conduct tests. Given more time, I am confident we would have been able to complete our goal. Overall, we haven’t been able to come to the conclusions we intended in this paper. We lacked the time needed to fully develop testing applications and gain a full and complete comprehension of the Android Operating System. We were surprised to find that little can be done without permissions and that much of Android’s security relies on the user; however, we would have liked to be able to test more permissions.

8. Future Work

Further research would open up opportunities for beta testing applications, attempting to bypass Bouncer or the Sandboxing of applications, developing more malicious code with as little permission as possible or even establishing root permissions through an application without the user’s knowledge. It is our intention that future research can be done from the research we have compiled and conducted in this paper in order to better understand the dangers and prevention systems of mobile malware.
APPENDIX A. Android Test Code

* Copyright (C) 2007 The Android Open Source Project
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and limitations under the Li-
* cense.
*/

package com.example.android.snake;
import android.app.Activity;
import android.os.Bundle;
import android.view.Window;
import android.widget.TextView;
/**
 * Snake: a simple game that everyone can enjoy.
 * This is an implementation of the classic Game "Snake", in which you control a
 * serpent roaming around the garden looking for apples. Be careful, though,
 * because when you catch one, not only will you become longer, but you'll move
 * faster. Running into yourself or the walls will end the game.
 */
public class Snake extends Activity {
    private SnakeView mSnakeView;
    private static String ICICLE_KEY = "snake-view";
    /**
     * Called when Activity is first created. Turns off the title bar, sets up
     * the content views, and fires up the SnakeView.
     */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.snake_layout);
        mSnakeView = (SnakeView) findViewById(R.id.snake);
        mSnakeView.setTextView((TextView) findViewById(R.id.text));
        if (savedInstanceState == null) {
            // We were just launched
            mSnakeView.setMode(SnakeView.READY);
        } else {
            // We are being restored
            Bundle map = savedInstanceState.getBundle(ICICLE_KEY);
            if (map != null) {
                mSnakeView.restoreState(map);
            } else {
                mSnakeView.setMode(SnakeView.PAUSE);
            }
        }
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Pause the game along with the activity
        mSnakeView.setMode(SnakeView.PAUSE);
    }
    @Override
    public void onSaveInstanceState(Bundle outState) {
        }
    }
}
// Store the game state
outState.putBundle(ICICLE_KEY, mSnakeView.saveState());
}

* Copyright (C) 2007 The Android Open Source Project
* Licensed under the Apache License, Version 2.0 (the "License");
* you may not use this file except in compliance with the License.
* You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0
* Unless required by applicable law or agreed to in writing, software
* distributed under the License is distributed on an "AS IS" BASIS,
* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
* See the License for the specific language governing permissions and
* limitations under the License.
*/
package com.example.android.snake;
import java.util.ArrayList;
import java.util.Random;
import android.content.Context;
import android.content.res.Resources;
import android.os.Handler;
import android.os.Message;
import android.util.AttributeSet;
import android.os.Bundle;
import android.util.Log;
import android.view.KeyEvent;
import android.view.View;
import android.widget.TextView;

/**
 * SnakeView: implementation of a simple game of Snake
 */
public class SnakeView extends TileView {
    private static final String TAG = "SnakeView";
    /**
     * Current mode of application: READY to run, RUNNING, or you have already
     * lost. static final ints are used instead of an enum for performance
     * reasons.
     */
    private int mMode = READY;
    public static final int PAUSE = 0;
    public static final int READY = 1;
    public static final int RUNNING = 2;
    public static final int LOSE = 3;
    /**
     * Current direction the snake is headed.
     */
    private int mDirection = NORTH;
    private int mNextDirection = NORTH;
    private static final int NORTH = 1;
    private static final int SOUTH = 2;
    private static final int EAST = 3;
    private static final int WEST = 4;
    /**
     * Labels for the drawables that will be loaded into the TileView class
     */
    private static final int RED_STAR = 1;
    private static final int YELLOW_STAR = 2;
    private static final int GREEN_STAR = 3;
    /**
     * mScore: used to track the number of apples captured
     * mMmoveDelay: number of
     * milliseconds between snake movements. This will decrease as apples are
     * captured.
     */
private long mScore = 0;
private long mMoveDelay = 600;
/**
 * mLastMove: tracks the absolute time when the snake last moved, and is used
 * to determine if a move should be made based on mMoveDelay.
 */
private long mLastMove;
/**
 * mStatusText: text shows to the user in some run states
 */
private TextView mStatusText;
/**
 * mSnakeTrail: a list of Coordinates that make up the snake's body
 * mAppleList: the secret location of the juicy apples the snake craves.
 */
private ArrayList<Coordinate> mSnakeTrail = new ArrayList<Coordinate>();
private ArrayList<Coordinate> mAppleList = new ArrayList<Coordinate>();
/**
 * Everyone needs a little randomness in their life
 */
private static final Random RNG = new Random();
/**
 * Create a simple handler that we can use to cause animation to happen. We
 * set ourselves as a target and we can use the sleep() function to cause an update/invalidate to occur at a later date.
 */
private RefreshHandler mRedrawHandler = new RefreshHandler();
class RefreshHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        SnakeView.this.update();
        SnakeView.this.invalidate();
    }
    public void sleep(long delayMillis) {
        this.removeMessages(0);
        sendMessageDelayed(obtainMessage(0), delayMillis);
    }
};
/**
 * Constructs a SnakeView based on inflation from XML
 * @param context
 * @param attrs
 */
public SnakeView(Context context, AttributeSet attrs) {
    super(context, attrs);
    initSnakeView();
}
public SnakeView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    initSnakeView();
}
private void initSnakeView() {
    setFocusable(true);
    Resources r = this.getContext().getResources();
    resetTiles(4);
    loadTile(REDD_STAR, r.getDrawable(R.drawable.redstar));
    loadTile(YELLOW_STAR, r.getDrawable(R.drawable.yellowstar));
    loadTile(GREEN_STAR, r.getDrawable(R.drawable.greenstar));
    }
private void initNewGame() {
    mSnakeTrail.clear();
    mAppleList.clear();
// For now we're just going to load up a short default eastbound snake
// that's just turned north
mSnakeTrail.add(new Coordinate(7, 7));
mSnakeTrail.add(new Coordinate(6, 7));
mSnakeTrail.add(new Coordinate(5, 7));
mSnakeTrail.add(new Coordinate(4, 7));
mSnakeTrail.add(new Coordinate(3, 7));
mSnakeTrail.add(new Coordinate(2, 7));
mNextDirection = NORTH;
// Two apples to start with
addRandomApple();
addRandomApple();
mMoveDelay = 600;
mScore = 0;
}
/**
* Given a ArrayList of coordinates, we need to flatten them into an array of
* ints before we can stuff them into a map for flattening and storage.
* @param cvec : a ArrayList of Coordinate objects
* @return : a simple array containing the x/y values of the coordinates
* as [x1,y1,x2,y2,x3,y3...]
*/
private int[] coordArrayListToArray(ArrayList<Coordinate> cvec) {
    int count = cvec.size();
    int[] rawArray = new int[count * 2];
    for (int index = 0; index < count; index++) {
        Coordinate c = cvec.get(index);
        rawArray[2 * index] = c.x;
        rawArray[2 * index + 1] = c.y;
    }
    return rawArray;
}
/**
* Save game state so that the user does not lose anything
* if the game process is killed while we are in the
* background.
* @return a Bundle with this view's state
*/
public Bundle saveState() {
    Bundle map = new Bundle();
    map.putIntArray("mAppleList", coordArrayListToArray(mAppleList));
    map.putInt("mDirection", Integer.valueOf(mDirection));
    map.putInt("mNextDirection", Integer.valueOf(mNextDirection));
    map.putLong("mMoveDelay", Long.valueOf(mMoveDelay));
    map.putLong("mScore", Long.valueOf(mScore));
    map.putIntArray("mSnakeTrail", coordArrayListToArray(mSnakeTrail));
    return map;
}
/**
* Given a flattened array of ordinate pairs, we reconstitute them into a
* ArrayList of Coordinate objects
* @param rawArray : [x1,y1,x2,y2,...]
* @return a ArrayList of Coordinates
*/
private ArrayList<Coordinate> coordArrayToArrayList(int[] rawArray) {
    ArrayList<Coordinate> coordArrayList = new ArrayList<Coordinate>();
    int coordCount = rawArray.length;
    for (int index = 0; index < coordCount; index += 2) {
        Coordinate c = new Coordinate(rawArray[index], rawArray[index + 1]);
        coordArrayList.add(c);
    }
    return coordArrayList;
public void restoreState(Bundle icicle) {
    setMode(PAUSE);
    mAppleList = coordArrayToArrayList(icicle.getIntArray("mAppleList"));
    mDirection = icicle.getInt("mDirection");
    mNextDirection = icicle.getInt("mNextDirection");
    mMoveDelay = icicle.getLong("mMoveDelay");
    mScore = icicle.getLong("mScore");
    mSnakeTrail = coordArrayToArrayList(icicle.getIntArray("mSnakeTrail"));
}

/**
 * handles key events in the game. Update the direction our snake is traveling
 * based on the DPAD. Ignore events that would cause the snake to immediately
 * turn back on itself.
 * (non-Javadoc)
 * @see android.view.View#onKeyDown(int, android.os.KeyEvent)
 */
@Override
public boolean onKeyDown(int keyCode, KeyEvent msg) {
    if (keyCode == KeyEvent.KEYCODE_DPAD_UP) {
        if (mMode == READY | mMode == LOSE) {
            /*
            * At the beginning of the game, or the end of a previous one,
            * we should start a new game.
            */
            initNewGame();
            setMode(RUNNING);
            update();
            return (true);
        }
        if (mMode == PAUSE) {
            /*
            * If the game is merely paused, we should just continue where
            * we left off.
            */
            setMode(RUNNING);
            update();
            return (true);
        }
        if (mDirection != SOUTH) {
            mNextDirection = NORTH;
        }
        return (true);
    }
    if (keyCode == KeyEvent.KEYCODE_DPAD_DOWN) {
        if (mDirection != NORTH) {
            mNextDirection = SOUTH;
        }
        return (true);
    }
    if (keyCode == KeyEvent.KEYCODE_DPAD_LEFT) {
        if (mDirection != EAST) {
            mNextDirection = WEST;
        }
        return (true);
    }
    if (keyCode == KeyEvent.KEYCODE_DPAD_RIGHT) {
        if (mDirection != WEST) {
            mNextDirection = EAST;
        }
        return (true);
    }
    return (false);
}
mNextDirection = EAST;
}
return (true);
}
return super.onKeyDown(keyCode, msg);
/**
 * Sets the TextView that will be used to give information (such as "Game
 * Over" to the user.
 * @param newView
 */
public void setTextView(TextView newView) {
    mStatusText = newView;
}
/**
 * Updates the current mode of the application (RUNNING or PAUSED or the like)
 * as well as sets the visibility of textview for notification
 * @param newMode
 */
public void setMode(int newMode) {
    int oldMode = mMode;
    mMode = newMode;
    if (newMode == RUNNING & oldMode != RUNNING) {
        mStatusText.setVisibility(View.INVISIBLE);
        update();
        return;
    }
    Resources res = getContext().getResources();
    CharSequence str = "";
    if (newMode == PAUSE) {
        str = res.getText(R.string.mode_pause);
    }
    if (newMode == READY) {
        str = res.getText(R.string.mode_ready);
    }
    if (newMode == LOSE) {
        str = res.getString(R.string.mode_lose_prefix) + mScore
            + res.getString(R.string.mode_lose_suffix);
    }
    mStatusText.setText(str);
    mStatusText.setVisibility(View.VISIBLE);
}
/**
 * Selects a random location within the garden that is not currently covered
 * by the snake. Currently _could_ go into an infinite loop if the snake
 * currently fills the garden, but we'll leave discovery of this prize to a
 * truly excellent snake-player.
 */
private void addRandomApple() {
    Coordinate newCoord = null;
    boolean found = false;
    while (!found) {
        // Choose a new location for our apple
        int newX = 1 + RNG.nextInt(mXTileCount - 2);
        int newY = 1 + RNG.nextInt(mYTileCount - 2);
        newCoord = new Coordinate(newX, newY);
        // Make sure it's not already under the snake
        boolean collision = false;
        int snakelength = mSnakeTrail.size();
        for (int index = 0; index < snakelength; index++) {
            if (mSnakeTrail.get(index).equals(newCoord)) {
                collision = true;
                break;
            }
        }
        if (!collision) {
            found = true;
        }
    }
}
if (newCoord == null) {
    Log.e(TAG, "Somehow ended up with a null newCoord!");
}

mAppleList.add(newCoord);

/**
 * Handles the basic update loop, checking to see if we are in the running
 * state, determining if a move should be made, updating the snake's location.
 */
public void update() {
    if (mMode == RUNNING) {
        long now = System.currentTimeMillis();
        if (now - mLastMove > mMoveDelay) {
            clearTiles();
            updateWalls();
            updateSnake();
            updateApples();
            mLastMove = now;
        }
        mRedrawHandler.sleep(mMoveDelay);
    }
}

/**
 * Draws some walls.
 */
private void updateWalls() {
    for (int x = 0; x < mXTileCount; x++) {
        setTile(GREEN_STAR, x, 0);
        setTile(GREEN_STAR, x, mYTileCount - 1);
    }
    for (int y = 1; y < mYTileCount - 1; y++) {
        setTile(GREEN_STAR, 0, y);
        setTile(GREEN_STAR, mXTileCount - 1, y);
    }
}

/**
 * Draws some apples.
 */
private void updateApples() {
    for (Coordinate c : mAppleList) {
        setTile(YELLOW_STAR, c.x, c.y);
    }
}

/**
 * Figure out which way the snake is going, see if he's run into anything (the
 * walls, himself, or an apple). If he's not going to die, we then add to the
 * front and subtract from the rear in order to simulate motion. If we want to
 * grow him, we don't subtract from the rear.
 */
private void updateSnake() {
    boolean growSnake = false;
    // grab the snake by the head
    Coordinate head = mSnakeTrail.get(0);
    Coordinate newHead = new Coordinate(1, 1);
    mDirection = mNextDirection;
switch (mDirection) {
    case EAST: {
        newHead = new Coordinate(head.x + 1, head.y);
        break;
    }
    case WEST: {
        newHead = new Coordinate(head.x - 1, head.y);
        break;
    }
    case NORTH: {
        newHead = new Coordinate(head.x, head.y - 1);
        break;
    }
    case SOUTH: {
        newHead = new Coordinate(head.x, head.y + 1);
        break;
    }
    // Collision detection
    // For now we have a 1-square wall around the entire arena
    if ((newHead.x < 1) || (newHead.y < 1) || (newHead.x > mXTileCount - 2) || (newHead.y > mYTileCount - 2)) {
        setMode(LOSE);
        return;
    }
    // Look for collisions with itself
    int snakeLength = mSnakeTrail.size();
    for (int snakeIndex = 0; snakeIndex < snakeLength; snakeIndex++) {
        Coordinate c = mSnakeTrail.get(snakeIndex);
        if (c.equals(newHead)) {
            setMode(LOSE);
            return;
        }
    }
    // Look for apples
    int appleCount = mAppleList.size();
    for (int appleIndex = 0; appleIndex < appleCount; appleIndex++) {
        Coordinate c = mAppleList.get(appleIndex);
        if (c.equals(newHead)) {
            mAppleList.remove(c);
            addRandomApple();
            mScore++;
            mMoveDelay *= 0.9;
            growSnake = true;
        }
    }
    // push a new head onto the ArrayList and pull off the tail
    mSnakeTrail.add(0, newHead);
    // except if we want the snake to grow
    if (!growSnake) {
        mSnakeTrail.remove(mSnakeTrail.size() - 1);
    }
    int index = 0;
    for (Coordinate c : mSnakeTrail) {
        if (index == 0) {
            setTile(YELLOW_STAR, c.x, c.y);
        } else {
            setTile(RED_STAR, c.x, c.y);
        }
        index++;
    }
/**
 * Simple class containing two integer values and a comparison function.
 * There's probably something I should use instead, but this was quick and
 * easy to build.
 */
private class Coordinate {
    public int x;
    public int y;
    public Coordinate(int newX, int newY) {
        x = newX;
        y = newY;
    }
    public boolean equals(Coordinate other) {
        if (x == other.x && y == other.y) {
            return true;
        }
        return false;
    }
    @Override
    public String toString() {
        return "Coordinate: [" + x + "," + y + "]";
    }
}

package com.example.android.snake;
import android.content.Context;
import android.content.res.TypedArray;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.drawable.Drawable;
import android.util.AttributeSet;
import android.view.View;

/**
 * TileView: a View-variant designed for handling arrays of "icons" or other
 * drawables.
 */
public class TileView extends View {
    /**
     * Parameters controlling the size of the tiles and their range within view.
     * Width/Height are in pixels, and Drawables will be scaled to fit to these
     * dimensions. X/Y Tile Counts are the number of tiles that will be drawn.
     */
    protected static int mTileSize;
    protected static int mXTileCount;
    protected static int mYTileCount;
    private static int mXOffset;
    private static int mYOffset;
    /**
     * A hash that maps integer handles specified by the subclasser to the
     * drawable that will be used for that reference
     */
private Bitmap[] mTileArray;
/**
 * A two-dimensional array of integers in which the number represents the
 * index of the tile that should be drawn at that locations
 */
private int[][] mTileGrid;
private final Paint mPaint = new Paint();
public TileView(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);
    TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.TileView);
    mTileSize = a.getInt(R.styleable.TileView_tileSize, 12);
    a.recycle();
}
public TileView(Context context, AttributeSet attrs) {
    super(context, attrs);
    TypedArray a = context.obtainStyledAttributes(attrs, R.styleable.TileView);
    mTileSize = a.getInt(R.styleable.TileView_tileSize, 12);
    a.recycle();
}
/**
 * Rests the internal array of Bitmaps used for drawing tiles, and
 * sets the maximum index of tiles to be inserted
 * @param tilecount
 */
public void resetTiles(int tilecount) {
    mTileArray = new Bitmap[tilecount];
}
@Override
protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    mXTileCount = (int) Math.floor(w / mTileSize);
    mYTileCount = (int) Math.floor(h / mTileSize);
    mXOffset = ((w - (mTileSize * mXTileCount)) / 2);
    mYOffset = ((h - (mTileSize * mYTileCount)) / 2);
    mTileGrid = new int[mXTileCount][mYTileCount];
    clearTiles();
}
/**
 * Function to set the specified Drawable as the tile for a particular integer key.
 * @param key
 * @param tile
 */
public void loadTile(int key, Drawable tile) {
    Bitmap bitmap = Bitmap.createBitmap(mTileSize, mTileSize, Bitmap.Config.ARGB_8888);
    Canvas canvas = new Canvas(bitmap);
    tile.setBounds(0, 0, mTileSize, mTileSize);
    tile.draw(canvas);
    mTileArray[key] = bitmap;
}
/**
 * Resets all tiles to 0 (empty)
 */
public void clearTiles() {
    for (int x = 0; x < mXTileCount; x++) {
        for (int y = 0; y < mYTileCount; y++) {
            setTile(0, x, y);
        }
    }
}
/**
 * Used to indicate that a particular tile (set with loadTile and referenced
 * by an integer) should be drawn at the given x/y coordinates during the next invalidate/draw cycle.
public void setTile(int tileIndex, int x, int y) {
    mTileGrid[x][y] = tileIndex;
}

@Override
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    for (int x = 0; x < mXTileCount; x += 1) {
        for (int y = 0; y < mYTileCount; y += 1) {
            if (mTileGrid[x][y] > 0) {
                canvas.drawBitmap(mTileArray[mTileGrid[x][y]],
                                   mXOffset + x * mTileSize,
                                   mYOffset + y * mTileSize,
                                   mPaint);
            }
        }
    }
}

/* AUTO-GENERATED FILE. DO NOT MODIFY.
* This class was automatically generated by the aapt tool from the resource data it found. It
* should not be modified by hand.
*/
package com.example.android.snake;
public final class R {
    public static final class attr {
        /** <p>Must be an integer value, such as "<code>100</code>".</p>
<code>
<p>This may also be a reference to a resource (in the form "<code>@[<i>package</i>:]<i>type</i>:<i>name</i></code>") or
theme attribute (in the form "<code>?key=[<i>package</i>:]<i>type</i>:<i>name</i></code>")
containing a value of this type.</p>

    *
    public static final int tileSize=0x7f010000;
    }
    public static final class drawable {
        public static final int greenstar=0x7f020000;
        public static final int redstar=0x7f020001;
        public static final int yellowstar=0x7f020002;
    }
    public static final class id {
        public static final int snake=0x7f050000;
        public static final int text=0x7f050001;
    }
    public static final class layout {
        public static final int snake_layout=0x7f030000;
    }
    public static final class string {
        public static final int mode_lose_prefix=0x7f040002;
        public static final int mode_lose_suffix=0x7f040003;
        public static final int mode_pause=0x7f040001;
        public static final int mode_ready=0x7f040000;
        public static final int snake_layout_text_text=0x7f040004;
    }
    public static final class styleable {
        /** Attributes that can be used with a TileView.
<p>Includes the following attributes:</p>
<table>
<colgroup align="left" />
*/
"}
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.example.android.snake:tileSize</td>
<td>0x7f010000</td>
</tr>
</tbody>
</table>

```
public static final int[] TileView = {
    0x7f010000
};
/**
* This symbol is the offset where the {link com.example.android.snake.R.attr#tileSize}
attribute's value can be found in the {link #TileView} array.
* Must be an integer value, such as "<code>100</code>".
* This may also be a reference to a resource (in the form
"<code>@{i}package:i:|i|type|i|name|</code>) or
theme attribute (in the form
"<code>?|i|package:i:|i|type:i|name|</code>
containing a value of this type.
* @attr name android:tileSize
*/
public static final int TileView_tileSize = 0;
```

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.snake">
    <application android:label="Snake on a Phone">
        <activity android:name="Snake"
            android:theme="@android:style/Theme.NoTitleBar"
            android:screenOrientation="portrait"
            android:configChanges="keyboardHidden|orientation">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```


