

A Security Analysis of Botting in Video Games

By Liam Gomez, Zach Mekaelian, Nicholas Malamud, Young Taek Oh

Abstract

In today's day and age, online video games have been a premier target of bots. In this study we analyze the current tactics that game developers use to combat botting in video games, and implement an experiment of our own in order to find the best ways to combat botting. Using an adversarial approach we identified how botting scripts overcome traditional prevention methods and how game developers can potentially secure their games in ways that do not ruin the experience of the player. We found that by utilizing the movement of a player's physical mouse we can distinguish their movement from that of a bot. This is accomplished by identifying that inputs processed within the Unity Engine have special designations as to which device they originate from whether it be virtual or physical. In doing this we are able to determine which inputs come from a bot, and which come from a human. The discovery made by this showcases the potential for future work in studying new ways for how physical devices can be used as CAPTCHAs.

Introduction

Nowadays online video games are plagued with players that are not real humans, but in fact scripts meant to play the game in an efficient manner. These bots are often incredibly harmful to the ecosystems of these online games and sometimes are enough to bring on the downfall of a game. Over time, bots have become increasingly more sophisticated. Game developers cannot keep up with the pace at which bots improve, leading to more bots than players appearing in these games, which leads to the downfall of many modern games. It is for this reason that we plan to investigate this issue. The purpose of this study is to analyze the behavior of botting scripts in video games, how they work, the methods on how to prevent them, and how they affect games. In this project we study the numerous tactics that bots employ to circumvent anti-cheating systems built into video games.

In addition to researching existing bot scripts and anti-cheat software, we implemented an experiment utilizing an adversarial approach. In our experiment we created a simple Unity game that allowed us to experiment with potential anti-cheat measures, botting detection systems, and botting scripts in a controlled environment. This experiment was designed to be adversarial as we divided our group into two separate teams. One team was responsible for developing the game and its security systems. The other team was responsible for developing botting scripts that would attempt to bypass the security of the game. The game consists of a simple two dimensional screen where the player clicks on a number of squares that can change colors and earns a score for doing so. We aimed to keep this game as simple as possible as the game itself is not the primary subject of this project. The specific strategies we utilized for the bots involved external input scripts and code injection. Security features for the game and new features for bot scripts were implemented over multiple iterations in order to emulate the cyclical process of how security features are implemented in games and circumvented over time.

To supplement our experiment we also provide a general overview of some of the implications of implementing more extreme security measures in video games. We investigate existing security measures found in popular online games that turned out to be more harmful

than beneficial to the games that they were designed for. The end goal of this project is to provide an overall analysis of our findings from both our experiment and our research done about security measures in other games. We hope to find a solution to botting in video games that does not negatively affect the players of the game nor the type of content that the game provides. Our primary hypothesis is that by analyzing the movement of a player through their actions based on input we can accurately determine the difference between a human player and a bot without needing to disrupt the gameplay itself.

Literature Review

Extant research shows that dealing with bots is a rather sophisticated task which involves not one, but many layers of security. In the study "Preventing Bots from Playing Online Games" by Philippe Golle and Nicolas Ducheneaut many strategies are discussed for preventing bots. "The problem of bots is not a superficial symptom of the limitations of existing games. Rather, it is an inherent consequence of the fact that repetition is a fundamental component of games," (Golle & Ducheneaut, 2005). Bots are designed to thrive in repetitive environments and this paper discusses the strategies needed to deal with addressing this specific flaw in video games. The study outlines many approaches to dealing with bots in video games and which ones are effective. The premise of their study is based around the idea of implementing CAPTCHAs or reverse turing tests into the gameplay of a game. The study found two key flaws with utilizing generic CAPTCHAs alongside video games: CAPTCHAs are intrusive and they can be outsourced to real people to be solved. They found that the best approach is typically to build the game itself as if it were a CAPTCHA. They also discussed how strong social elements present within games often make for great CAPTCHAs. The study ultimately concludes though that those implementations are not without their issues and that physical CAPTCHAs are often the strongest out of every option. Physical CAPTCHAs meaning that a player's input device, such as a controller, is made tamper proof in that inputs cannot be spoofed and must be verified before being sent (Golle & Ducheneaut, 2005).

The article "Online game bot detection based on party-play log analysis" by Ah Reum Kang, Jiyoung Woo, Juyong Park, and Huy Kang Kim also discusses botting in MMORPGs. The article goes into depth on several bot detection methods that are popular in the game industry, categorized into client side, network side, and server side detection. Client side detection methods revolve around attempting to detect any botting happening on the game client. One way to detect client side botting is through the use of anti-cheat and security solutions for games. Another way to detect client side botting is by designing the game in a way that can catch bots. For example, creating an invisible object that only a program interacting with the engine could detect would easily rule out any legitimate players. Network side monitoring methods revolve around detecting any network activity that the bots create, or constantly changing the network protocols. Traffic monitoring tends to have a high rate of false positives, and network protocol changes tend to have a high cost in terms of both user experience and server cost for encryption and decryption in real-time. Server side detection methods try to detect bots by looking at the activity being logged in the server and identifying bot-specific patterns (Kang et al., 2013).

In "How to break a CAPTCHA system in 15 minutes with Machine Learning" by Adam Geitgey, we can see just how simple it is to break one of the main methods companies use to

catch bots: CAPTCHA. The article reveals that using basic machine training and openly available code libraries, a program can start to solve simple text CAPTCHAs on its own. While the author discourages the use of this CAPTCHA solving bot on real websites, he also shows just how easy it is to make and even provides the resources he used to make it. This article perfectly demonstrates the arms race that happens between bot and game developers. With enough time and effort, bot developers are seemingly able to overcome any challenges that are thrown at them (Geitgey, 2019).

Out of the mentioned research, the work by Golle & Ducheneaut (2005) is of particular interest to us. Their study discusses the implementation of hardware level CAPTCHAs in order to valid user input. Our experiment hopes to expand upon this idea of input validation through software level solutions for existing devices rather than building brand new tamper proof devices. We also chose to focus on mouse movement specifically rather than input from all devices.

Method

In our experiment we sought to create a secure environment against botting by taking an adversarial approach. In this approach we had multiple iterations of our development cycle which alternated between security testing, and development time to improve and add new features. This process is outlined by Figure 1 and represents the cyclical nature of development that both bot and game developers encounter in the real world.

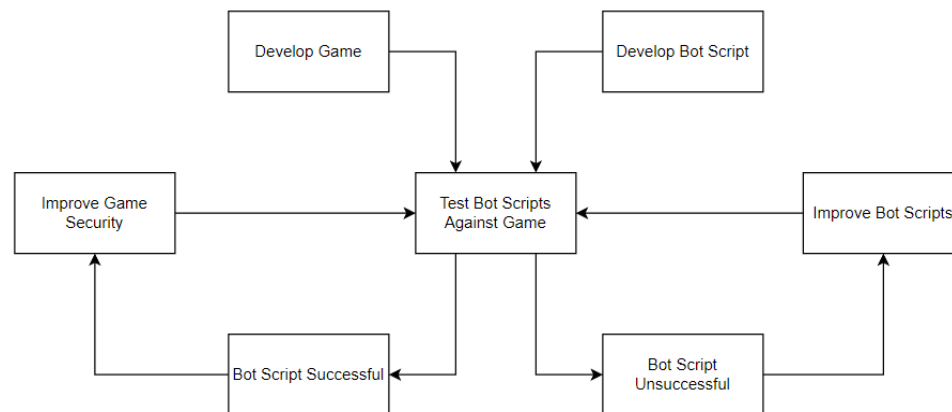
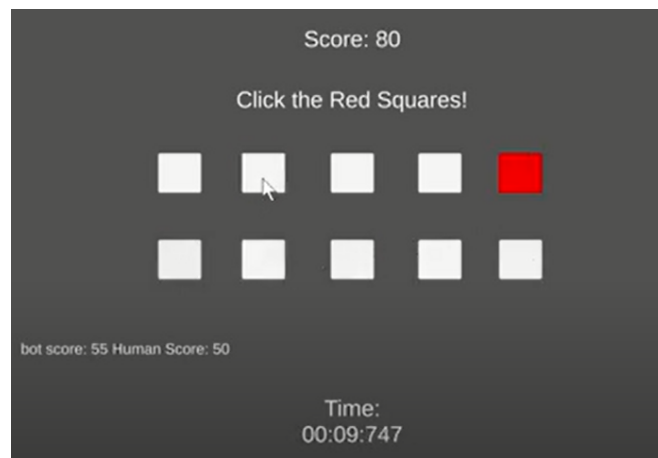


Figure 1

Implementation of the Game

We implemented a simple game within the Unity engine in which the goal is to click on a series of red squares as fast as possible. The game, as shown in Figure 2, consists of a two-dimensional plane with a gray background, a grid of white and red squares, a timer, a game-score indicator, and a human-bot score display. The game begins with a single red square placed within the grid of white squares. Once the first red square is clicked it will become a white square and another square on the grid will become red. The timer then starts counting at this point and will continue to count until 10 red squares have been clicked. Each time a red square is clicked the score increases by 10. Once the score has reached 100 the game will end and the user is prompted with an ending screen and their time. This game mechanic was specifically

chosen for the experiment as one of the most common activities that a bot will perform in video games is moving the mouse and clicking on objects.



Gameplay Deterrents

Following the creation of the gameplay mechanics, we implemented a number of deterrents alongside the game which could be used to enhance player experience and discourage the developers of botting scripts. The first deterrent which ties into the gameplay itself is randomization. Each time the player begins the game, the grid of squares is “seeded” and randomized to produce a unique pattern in which the red squares appear every time. This randomization prevents botting scripts from following a memorized pattern every time that they play the game. In addition to introducing difficulties to bots, this mechanic also enhances the player experience by providing variety to the gameplay.

Furthermore, a pseudo banning feature was implemented into the game in order to provide a means of identifying when bots were detected. Normally a ban would be considered a deterrent in real world scenarios. In our experiment, however, the pseudo banning system does nothing more than redirect the user to a screen that proclaims they are banned. During our experiment we did not see the need to fully implement this already well established means of deterrent against bots as our focus was on detection and prevention from the technical level and not the social level. The threat of a ban is meaningless in our experiment and would have only slowed down the testing of our security features had it been fully implemented.

Online Services and CAPTCHA

Our game was implemented to be an online only experience in order to create a more realistic environment in which the bots will operate. Features such as account creation, user authentication, and login screens were implemented in order to simulate this. Authentication and our back-end services were handled by Microsoft’s PlayFab. In order to play the game players are required to create an account and login to said account before they can play the game. A valid email address and password are required for account creation. The page for account creation can be seen in Figure 3. This process all happens within a game launcher which was implemented using a MAUI and C#. Upon a successful authentication, the game process is then launched and a session token is then passed through a command-line argument. This session

token is then validated within the game and if not valid then the user will be unable to play the game. Requiring the player to be logged in not only presents game developers with the ability to uniquely identify players, but also allows them to control who is able to play their game and when. Additionally it provides a large barrier to entry for bots as they have to create an account to play the game which requires a valid email address.


We also implemented a CAPTCHA within our account creation process in order to further mimic real world scenarios. We ultimately faced many challenges with this though as we soon discovered CAPTCHAs were not designed to be implemented for our use cases. The initial idea was to place the account creation system entirely within the game itself. The issue with this, however, was that it would require a completely custom solution and implementation as no existing CAPTCHA provides had APIs that were generic enough to hook into C# code for use in the Unity game engine. An alternative that we settled on was to use a game launcher and put the process there, but this also proved to have similar issues. Many CAPTCHAs are designed to be specifically utilized for websites only and require a website domain to be linked to them. Requiring the linkage of a website domain to our application would present a multitude of challenges in which the solutions were not ideal. Fortunately we found a free and open source CAPTCHA system which runs locally known as BlazorCaptcha that is designed for Blazor applications. With MAUI applications being built upon ASP .NET and Blazor this was the ideal solution for our game launcher.

The CAPTCHA, showcased in Figure 3, was text based and provided a number of features that make text extraction exceedingly difficult. Each image generated provided unique character combinations with no repetitions. Additionally, many features that make image recognition difficult were present such as each character being a variety of different colors, random lines being placed throughout the image, and characters being only partially visible. The major downside to this CAPTCHA system though is that verification is done on device which provides some possibility for tampering despite its robust security features. Having a CAPTCHA in the traditional sense is only a small part of what our experiment hoped to accomplish.

Email:

Username:

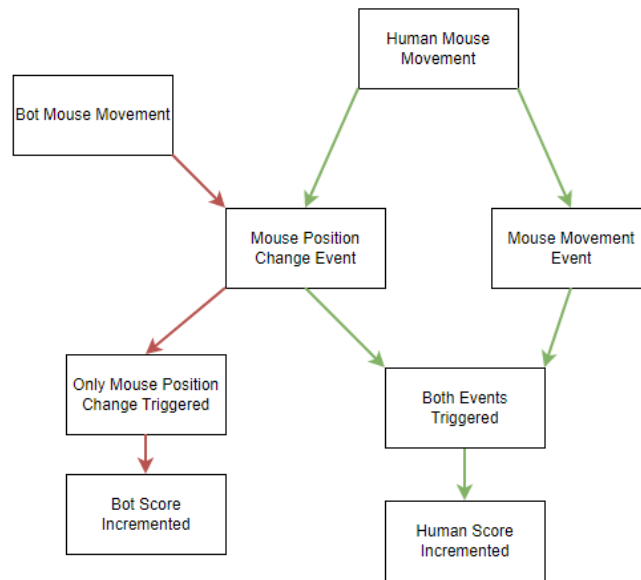
Password:



Enter Captcha:

e2\$U9

Botting Detection



The true objective of this project is our implementation of botting detection via physical input validation. In order to accomplish this objective we utilized a clever feature of Unity's input system. The input system in Unity is responsible for listening to and managing a plethora of different devices. This gives developers the ability to effectively distinguish between the input of different virtual and physical devices. With this knowledge in mind we set up input bindings for a physical mouse and began listening to input from these types of devices. Using these new input bindings we set up two different listener events within Unity. One of these events was to listen to mouse movement changes, and the other was to listen for mouse position changes. The idea here is that since the mouse position is not tied to any one device it will be picked up regardless of what device is in control of the cursor, but the mouse movement event will only be picked up by the device it is linked to. We hoped that by cross-referencing these two events we would be able to provide input validation for physical devices. Figure 4 outlines the detailed process of this input validation. Any time that a mouse position change occurs without a mouse movement event we could flag that specific incident as being suspicious. This method of detection would not be completely perfect in acting as validation though.

Instead of outright banning players for a single instance of cheating we would need to develop a scoring system to handle their suspicion. The need for a scoring system is due to the nature of using mouse position changes without movement as a basis for suspicion. There are scenarios in which false positives for suspicious activity can be created. One such scenario which was discovered early on into the experimentation is tabbing out of the game. Any time that the application loses focus it stops retaining information about the mouse and its movement. This means that when the player clicks back into the game window its position will have "teleported" and the game will flag this as suspicious activity. Therefore, since the need to account for false positives exists we needed to provide a system of leniency to the players.

The human-bot scoring system consists of two variables which assign a human and a bot score to players. Human score is incremented whenever input is validated against the physical device and bot score is incremented whenever invalid input is detected. Since human movement produces move events than invalid movement does, we needed to weigh the two

scores differently. The bot score was multiplied by a factor of 9 in order to account for the discrepancy between the two. We also accounted for the event of false positives by withholding judgment from the player until the game has ended. After the gameplay session has ended their human and bot scores are compared and players are banned if their bot score is greater than their human score. If the human score is greater then they are instead sent to the games end screen and their time is displayed to them. This system is aimed at targeting bots that utilize external input scripts as their primary method of interaction with the game. Botting scripts that fall under the category of code injection require a different method of identification.

Anti-Cheat

In order to identify bots that made use of tactics such as code injection and memory manipulation we needed an implementation that would target all generalized forms of client-side cheating. To accomplish this we utilized a mix of secure programming practices and specific tools to protect against data and memory manipulation.

Our code design philosophy within this project was to keep data as private as possible. Most data members within the project were private variables and provided very little opportunity for interaction outside of the classes that they belonged to. The variables that did require being public, however, were ensured to only be things that are cosmetic in nature such as text that is displayed on the screen to the user. These practices ensure that anyone analyzing the source code will have a difficult time manipulating anything to their advantage with outside scripts.

On top of that, we utilized a tool for memory and data protection which was a free plugin from the Unity Asset store called Anti-Cheat Free by GuardingPearSoftware. This plugin provided a means to encapsulate data members within Unity and also provided encryption for the data members upon compilation. Additionally, the plugin also supplied a way to detect when memory was being manipulated. This came in the form of an event listener hook which when called provided a customizable outcome after detection. In our case we set up the event listener to take players straight to the ban screen if memory manipulation was detected. This not only prevented attackers from analyzing the code, but provided a deterrent against them as well.

Implementation of Botting Scripts

We decided to use the Python programming language, coupled with the PyCharm editor to develop and run the bot. For controlling the mouse, we decided to use the PythonAutoGUI library, because of its popularity and ease of use. While playing the game, we would run the script on the PyCharm editor in a separate window.

Identifying the Colored Squares

Our approach for identifying the colored squares involved taking a screenshot of the game then applying functions from the OpenCV library to filter out and identify the red. First we generated a picture in black and red, anything not red was turned black. Then we converted the image to black and white, with the square being white and everything else black. The black and white image was fed into a OpenCV library function that could identify the color difference between the black and white and generate points around the edge of the color contrast. We then took the average of the points to get the location of the square in the middle, then proceeded to move to the position and successfully click it. This method was very accurate in

identifying the red square. It was able to identify the squares in a reasonable time, although slower than a regular human, we found it was much faster than going through every pixel on the screen searching for a red one.

Organic Mouse Movement

The previous implementation of our bot only moved the mouse directly to the position. It would teleport without any movement in between. This is obviously a red flag, which would make the bot easily identifiable. Our plan to work around this problem was to generate a function that could simulate mouse movement to make it unrecognizable when compared to a human's mouse movement. To do this we analyzed what a human's mouse movement looks like, we noticed that a human can not move their mouse in a straight line to the point. Our bot would need to make a squiggly path to the button, as to throw off any identification looking for straight line movement. Another identifier is speed, a human moves their cursor at different speeds. We would need to implement a change of speed to our bot to throw off the detection of this. In order to achieve these aforesaid results, we came up with a method using interpolation, which involves moving to the target with many small steps. We would randomize the location of the point in each step then generate a new path to the destination. Additionally we lowered the randomness as the mouse got close to the target, as to replicate a human "zeroing in" on to the button as it got close. A mouse that moves erratically all the way to the button would not mirror a human's and would be identifiable. Although it mirrored the movement of a human pretty well, it was notably slower.

Attacking the CAPTCHA

For our attack on BlazorCaptcha. We tried to use the method listed out In the tutorial, "How to break a CAPTCHA system in 15 minutes with Machine Learning" by Adam Geitgey, which was mentioned in the literature review. The first thing we needed to do was collect a large dataset of Captcha images from BlazorCaptcha to feed into the machine learning program. We were able to take a screenshot of the region of the screen that showed the Captcha, then refresh to generate a new Captcha. Repeating this process, we were able to generate a large dataset of captchas. A portion of this dataset is showcased in Figure 5. We then tried to use the given code from the tutorial and apply it to our dataset. We noticed that the tutorial used older code which had functions that had been changed in the library, it initially generated many errors. After fixing up the code, it was unable to perform the first stage of separating the letters from the images, because of the lines going through the Captcha. The Captcha that was used in the tutorial was much simpler than ours, it was only black and white with 4 letters either alphabetical or digits. Ours came in many colors, had lines going through it, was 5 digits and involved punctuation. It was basically the evolved version of the previous Captcha. All these new additions made it exponentially harder to separate the characters and apply machine learning to them.



Exploitation via Code Injection

For our code injection attempt, we used a program known as SharpMonoInjector to inject an updated dll code into the Unity game. In order to do so, we first needed to break open the Assembly-CSharp.dll file that stores all the methods of the game. This is necessary in order to be able to write a program that will be able to exploit and overwrite methods from the game. After understanding how the clicker game worked, the next step was to set up a project in Visual Studio that has the dll files for the game and the Unity engine imported. Creating an injectable dll was a two step process. A loader file needed to be written that would then load the cheat file that was written. The loader file was very simple, needing a load and unload method, so that the modified dll file could be removed if necessary. The cheat file was where the bulk of testing was performed, with several different ways of overwriting the game methods attempted. However, due to proper class separation, no variables were able to be updated, nor were the bot team able to call methods to make the game break.

Results

The results of our experiment ultimately showcased that the security features implemented by the game team were effective in stopping botting scripts in their tracks. The bot was able to successfully identify and click on the colored squares, but the image processing involved with identifying the colored squares unfortunately brought on some unwanted side effects with the bots performance. Since the bot had to spend time to think about where the next colored square was located it often would produce slower times than human players. This meant that the randomization factor within the game proved to be an effective deterrent as now bots had a disadvantage when compared to real players. Additionally, despite its best efforts at mimicking human behavior through interpolated movement it was still detected by the game's

input validation system. It was noticed that when ran through the input validation system, the bots movements were not picked up by the on mouse movement events, but they were picked up by the mouse position change events as intended. This led to the bot being easily recognized and banned within all of the tests that we ran. Any bots made using the PythonAutoGUI library can most likely be recognized by the input not being picked up. Likewise, the system performed well in recognizing input from real humans. None of the human gameplay tests we performed resulted in a ban which shows promise for the system as a whole in regards to false positives.

In addition to the external botting scripts, all methods of code injection were proven to be unsuccessful. The application was deemed to be secure enough by our botting team that they were ultimately unable to decipher a way to hook into the code in any meaningful way. As a result, the listener for memory manipulation was not able to be tested.

The account creation process was also unable to be completed by the botting scripts due to the inclusion of the CAPTCHA. The CAPTCHA and its various methods for preventing image processing proved to be effective in confusing the machine algorithms that were applied to solve the CAPTCHAs. Ways to bypass the authentication system were also not found by the team responsible for code injection.

Conclusion

In the end, this experiment showed evidence towards our hypothesis being correct. It is indeed possible to find new ways of implementing botting detection without diminishing the player experience in the game. We can also implement these bot detecting systems in our game with little resources and with scripts hard coded into our game environment. We presented this in our experiment by demonstrating that bots are able to be differentiated from human players via the game's physical input validations. Additionally, we showcased how effective traditional CAPTCHAs are in keeping bots from differentiating characters via artificial intelligence and machine learning. By applying simple techniques such as distorted letters, punctuation, different colors and lines you can drastically decrease the chances and speed that a bot's machine learning algorithms can decipher captcha text. Botting is a serious issue in today's world which diminishes player experience and exploits online games for financial gain. We hope that in future studies we can learn how to combat these bots even more effectively and keep games entertaining and enjoyable for the majority of everyday consumers.

Discussion

Our project was limited by the platforms in which we could code our game, and ultimately the time allocated for us to finish our assignment. We chose to use the free game development tool Unity for this project which uses the C# coding language, but there are many other game development environments that we could have potentially utilized in the making of this project. Other game development environments use different coding languages and have different integrated tools which have the potential to help aid in detecting malicious botting behavior. Given more time and resources we could have developed more ways to detect bot activity and the ability to combat more advanced and sophisticated botting software. Our original goal before discovering input validation was to utilize machine learning to identify bots using their acceleration and path trajectories. This posed a number of challenges, however, as most of

the potential datasets we could have used were very biased towards the mouse movement of a single user. Initial training showed that the datasets would ultimately be unusable for our objective of distinguishing humans and bots. It was during our exploration of recording mouse movements within Unity that we discovered software based input validation was possible. With this discovery we decided to shift our focus with the experiment to input validation with mouse movement. We as a group would also like to test out other different physical mediums in which to detect bots. This includes all input devices other than mouse movement such as keyboard and game controller inputs. By doing this we can detect different bot software such as keystroke logging bots. Furthermore we would like to further investigate whether we can spoof our methods of physical input validation using other botting methods that may provide human interface device packets.

References

- CBS News. (2021, April 3). *Game Over: China arrests 10, says it's busted the world's biggest video game cheating ring*. *Www.cbsnews.com*.
<https://www.cbsnews.com/news/china-video-cheating-ring-hackers-arrested/>
- Geitgey, A. (2019, May 2). *How to break a CAPTCHA system in 15 minutes with Machine Learning*. Medium.
<https://medium.com/@ageitgey/how-to-break-a-captcha-system-in-15-minutes-with-machine-learning-dbebb035a710>
- Golle, P., & Ducheneaut, N. (2005). Preventing bots from playing online games. *Computers in Entertainment*, 3(3), 3. <https://doi.org/10.1145/1077246.1077255>
- Kang, A., Woo, J., Park, J., & Kim, H. (2013). Online game bot detection based on party-play log analysis. *Computers & Mathematics with Applications*, 65(9), 1384–1395.
<https://doi.org/10.1016/j.camwa.2012.01.034>